Summer Project Report 2023

~By Aditya Kishore (ENGS3249)

(Indian Institute of Science Education and Research, Bhopal)

Summer Research Fellowship Program 2023

Project Title:

"Process of Line Detection Using Embedded Systems" and "Road Sense: Intelligent Road Symbols Classification, Behaviour Cloning for Autonomous Driving Systems"

Supervisor and Guide:

Dr. Sumana Ghosh

Electronics and Communication Sciences Unit

Computer and Communication Sciences Division

INDIAN STATISTICAL INSTITUTE, KOLKATA

ABSTRACT:

- In many applications, notably those involving robotics and autonomous vehicles, line detection is essential. An innovative method for line detection using embedded systems is presented in this abstract. The suggested system accurately detects and tracks lines in real-time by utilizing the effectiveness and capability of embedded platforms. The hardware for the system comprises of a low-power embedded board that has Sensor module attachments. The embedded platform's optimized line detection algorithm is used to process the live feed of the surroundings that the sensors have captured. The algorithm uses methods to recognize and extract lines from the sense and feedback data that has been recorded.
- The identified lines can be used for different applications, such as lane keeping, obstacle detection, and path planning in autonomous cars. The embedded system's computing capability enables for real-time execution, enabling immediate line recognition and tracking in dynamic contexts.
- The experimental findings show how effective and efficient the suggested line-detecting technique is. Even in difficult settings, including fluctuating lighting, various road surfaces, and intricate road markings, it manages to recognize lines reliably. The system is a good fit for embedded platforms with limited resources because of its low power consumption and real-time performance.
- Whereas Classification of Road Symbols and Behavioural Cloning are the two important aspects in the field of autonomous driving Systems. Modern computer vision algorithms and machine learning strategies are used by the system to extract features from the collected images or video streams. In order to classify the road symbols seen in the scene, these features are then fed into a trained classification model, such as a convolutional neural network (CNN). The classification results give the vehicle's perception system important information that it can use to decide what to do and how to do it.
- The system includes behavioural cloning in addition to road symbol classification to improve the way the vehicle drives. The process of "behavioural cloning" is copying the behaviours of experts in comparable situations after learning from their examples. The system can learn the required driving behaviours and mimic them in actual driving scenarios by utilising deep learning techniques like recurrent neural networks (RNNs) or generative adversarial networks (GANs).
- The behavioural cloning module employs the learned driving policies in combination with input from the perception system to generate the correct control signals, such as steering, acceleration, and braking, to safely manoeuvre the vehicle. As a result, the autonomous car can traverse challenging road conditions, adhere to traffic laws, and react to changing circumstances using previously learnt behaviour.

- As a result, the integrated strategy of behavioural cloning and road symbol categorization described in this abstract offers a complete remedy for enhancing the perception and judgement capabilities of autonomous vehicles. Through this integration, they are better able to comprehend the state of the roads, identify traffic signs, and provide a reliable and effective embedded system solution for line detection. Its possible applications include robots and driverless vehicles, where precise and immediate line detection is necessary for secure and dependable navigation.

Introduction:

Autonomous line-detecting and robots have revolutionized the field of robotics by combining advanced sensors, intelligent algorithms, and precise control systems. These robots are designed to navigate complex environments, detect lines or paths, and avoid obstacles with minimal human intervention. They play a crucial role in various industries, from manufacturing and logistics to agriculture and surveillance. The ability of these robots to operate autonomously, making real-time decisions based on sensor feedback, offers numerous advantages in terms of efficiency, safety, and productivity. This Report will explore the key features and applications of autonomous line-detecting and obstacle-avoidance robots, highlighting their potential to transform industries and improve our everyday lives. These types of autonomous Robots can be used in solving prominent problems like detecting landmines etc., in defence fields which is very beneficial for armies because it reduces human casualties and can reach tight and unreachable places. Line follower actually senses the line and follows it. Though the idea sounds simple, with a little more development, robots similar to this are practically used in many applications like factory floor management robots or warehouse robots.

Understanding line Detection:

Line Detection: What is it?

The process of locating and extracting lines or boundaries from an image or video feed is referred to as "line detection." It is essential for giving machines the ability to detect and navigate their surroundings efficiently. An embedded system is a crucial part of autonomous cars and intelligent systems because it can collect important data about object boundaries, road lanes, and other significant aspects by recognising lines. whereas

Road-Sense: Behaviour Cloning for Autonomous Driving Systems and Intelligent Road Symbol Classification

Classifying intelligent road symbols

A crucial component of autonomous driving systems is the classification of intelligent road symbols. Advanced machine learning techniques are used by embedded systems to enable automobiles to comprehend and react appropriately to road signs and symbols. These algorithms successfully identify and categorise road symbols by utilising computer vision techniques and neural networks. Vehicles can make educated decisions based on the identified symbols by incorporating intelligent road symbol categorization into autonomous driving systems, assuring increased safety and effectiveness on the roads.

The Cloning of Behaviour for Autonomous Driving Systems

Another key idea in autonomous driving systems is behaviour cloning. Vehicles can learn from human experience and drive themselves by capturing human driver behaviour and teaching an embedded system to emulate it. This method involves gathering data from multiple sensors, including cameras, LiDAR, and GPS, and then utilising machine learning techniques to train the embedded system. The system gains the ability to translate sensor inputs to the proper control actions, allowing the vehicle to successfully imitate human driving behaviour.

Embedded System's Effect on Modern Applications:

By offering clever and effective solutions, embedded systems have revolutionised many different sectors. Let's investigate the effects of embedded systems in some important fields:

Autonomous Vehicles:

The topic of autonomous driving is quickly developing, and embedded technologies are essential to its development. Vehicles can properly see and comprehend their surroundings thanks to embedded systems, which include sensors, CPUs, and complex algorithms. By enabling autonomous vehicles to navigate roadways, identify impediments, and make judgements in real time, this technology helps to create safer and more effective transportation networks.

Robotics:

Robotic applications are centred on embedded systems, which enable robots to carry out difficult jobs precisely and effectively. Robots with embedded systems can execute complex activities, interact with the environment, and help humans in a variety of disciplines, from industrial automation to healthcare and exploration. Robots can detect their environment and complete tasks on their own thanks to embedded systems that make use of sensors, actuators, and clever algorithms.

Industrial Automation:

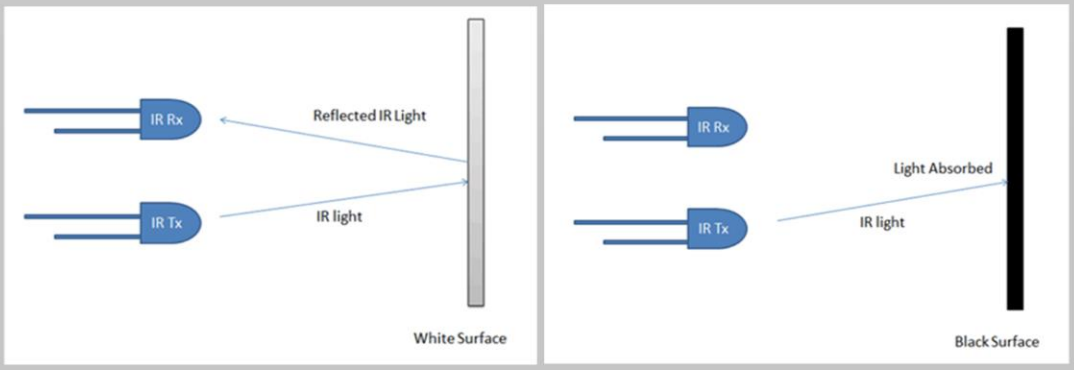
By making it possible for machinery, sensors, and control systems to be seamlessly integrated, embedded systems have revolutionised industrial automation. The productivity, accuracy, and safety of manufacturing processes are improved by embedded systems' precise control and monitoring capabilities. These solutions optimise processes, reduce downtime, and boost overall effectiveness across the board, from assembly lines to quality control, resulting in considerable cost savings and improved competitiveness.

And the Combination of all these three points can be implemented in real life with help of a Line-Following Robot(An embedded System).

Concepts of Line-Following Robot:

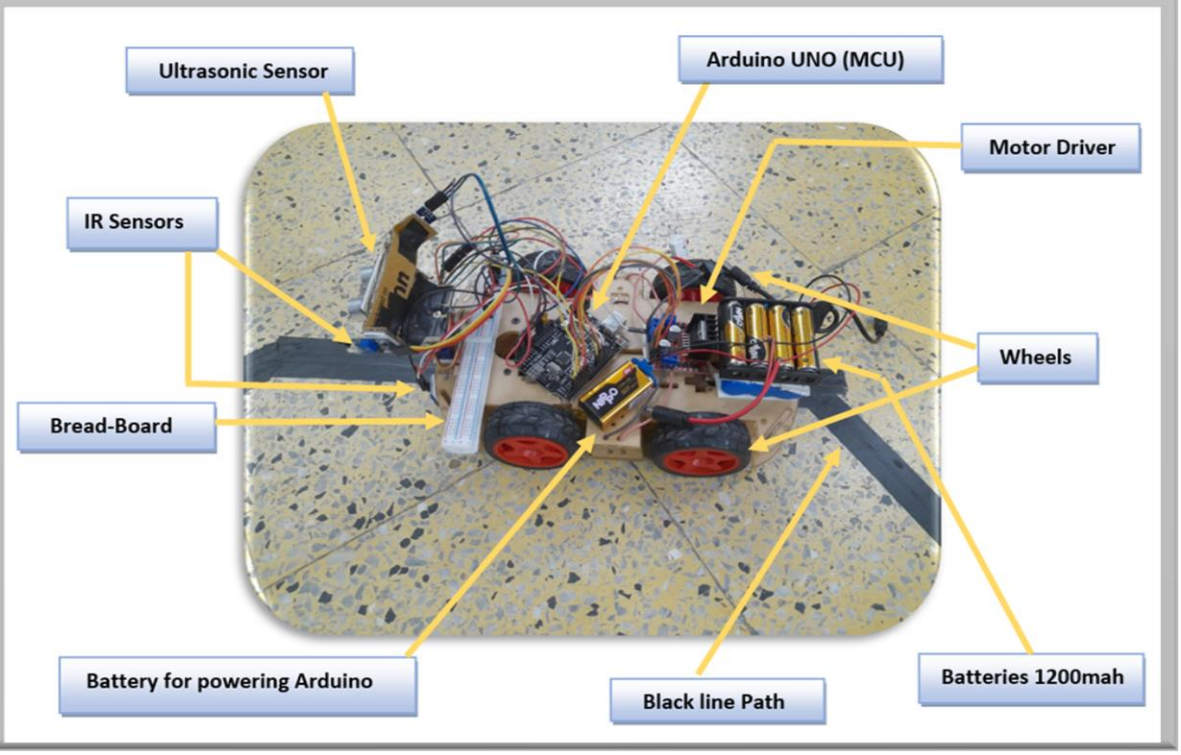
Ideas for Line Followers

Light is a factor in how a queue follower operates. Here, we make use of how light behaves on white and black surfaces. Light is almost entirely reflected when it hits a white surface and completely absorbed when it hits a black surface. A robot that follows lines is constructed using this property of light.



In this Project I have worked Upon Black Surface line detection.

In this Arduino based line follower robot, I have used IR Transmitters and IR receivers also called photodiodes. They are used for sending and receiving light. IR transmits infrared lights. When infrared rays falls on the white surface, it's reflected back and caught by photodiodes which generate some voltage changes. When IR light falls on a black surface, light is absorbed by the black surface and no rays are reflected back, thus photo diode does not receive any light or rays. Here in this Arduino line follower robot when the sensor senses white surface then Arduino gets 1 as input and when senses black line Arduino gets 0 as input.

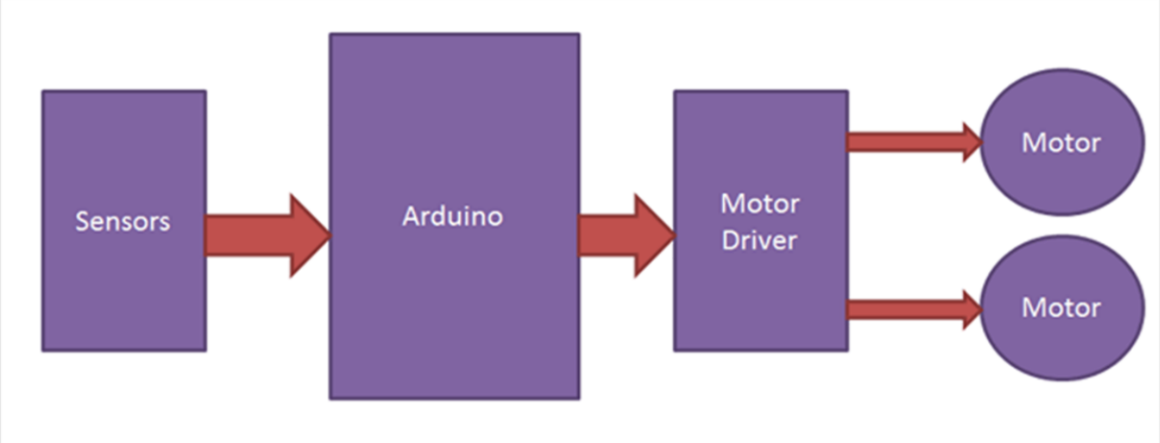


Components and Materials:

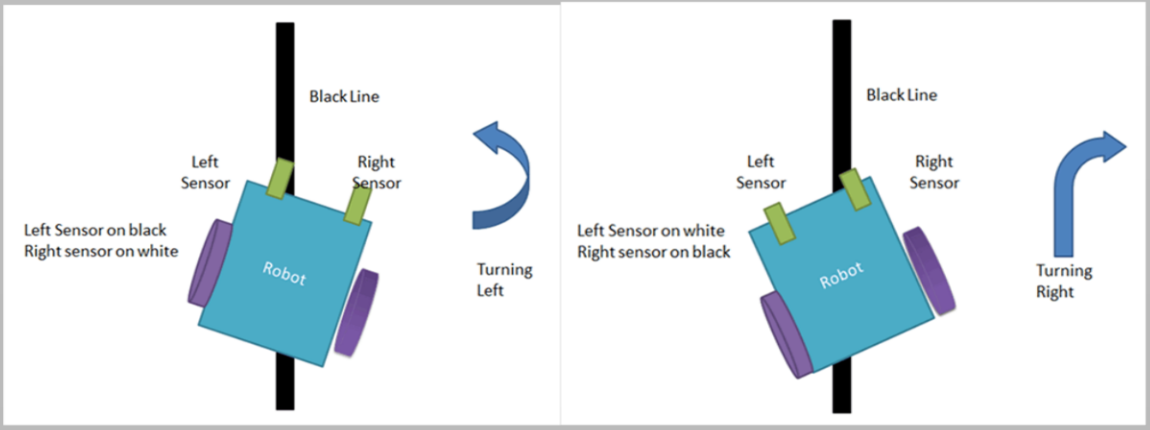
1. Robot chassis or platform
2. Motors and wheels for locomotion
3. Line detection sensors (e.g., infrared sensors, camera)
4. Microcontroller or development board (e.g., Arduino, Raspberry Pi)
5. Motor driver circuitry
6. Power source (e.g., batteries)
7. Supporting hardware (wires, connectors, screws)
8. Programming environment and software (e.g., Arduino IDE, Python)

Working of Line Follower Robot using Arduino:

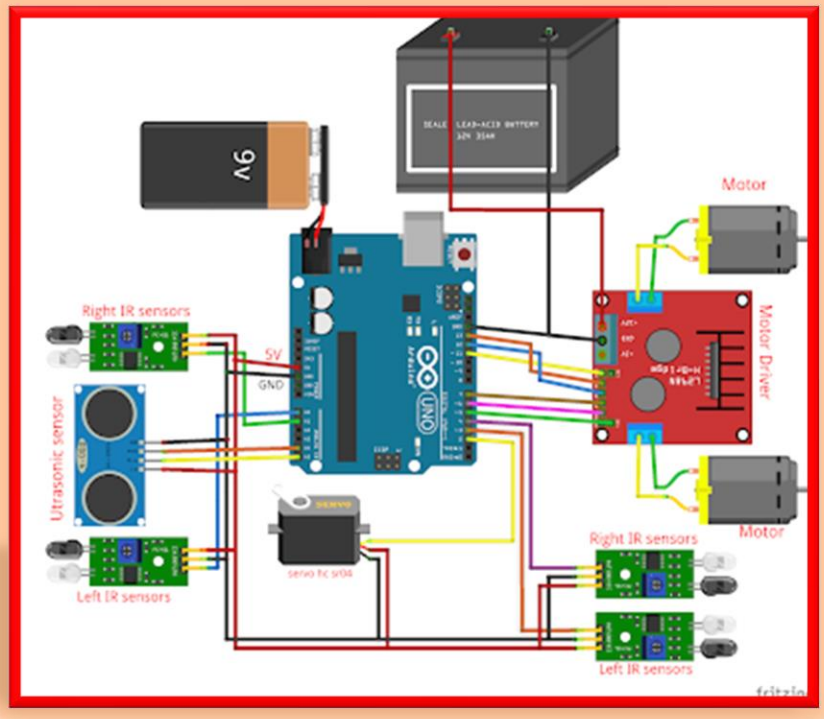
Building a Line follower robot using Arduino is interesting. The line follower robot senses a black line by using a sensor and then sends the signal to Arduino. Then Arduino drives the motor according to sensors' output.



Here in this project, we are using two IR sensor modules namely the left sensor and the right sensor. When both left and right sensor senses white then the robot moves forward.



Circuit Diagram:



Program Explanation:

line_follow | Arduino IDE 2.1.0

File Edit Sketch Tools Help

Arduino Uno

line_follow.ino

```
1   //motors
2   #define motora1 13
3   #define motorb1 12
4   #define motora2 7
5   #define motorb2 6
6   #define enA 11
7   #define enB 5
8
9   //ir sensors
10  #define LirSensor 3
11  #define RirSensor 4
12
13  int maxSpeed = 100;
14
15  int turnSpeed = 80;
16
17  void setup() {
18    Serial.begin(9600);
19    pinMode(motora1, OUTPUT);
20    pinMode(motorb1, OUTPUT);
21    pinMode(motora2, OUTPUT);
22    pinMode(motorb2, OUTPUT);
23
24    pinMode(enA, OUTPUT);
25    pinMode(enB, OUTPUT);
26
```

Defining functions in the code for various motors' output and inputs associated with their appropriate pin numbers of Motor Driver

Defining functions for left and right IR sensors at 3 and 4 pins of Arduino for Output pins of IR

Fixing speed for motors.

Creating a void setup for motors pins for output 1,2,3,4 for left and right motors along with enabling A and B pins of the Motor Driver. After connecting all inputs pin of the motor driver to Arduino (Microcontroller Unit).

line_follow | Arduino IDE 2.1.0

File Edit Sketch Tools Help

Arduino Uno

line_follow.ino

```
27    pinMode(LirSensor, INPUT);
28    pinMode(RirSensor, INPUT);
29  }
30
31  void loop() {
32    //for Line follow
33    int LEFT_SENSOR = digitalRead(LirSensor);
34    int RIGHT_SENSOR = digitalRead(RirSensor);
35
36    Serial.print("leftsenor: ");
37    Serial.print(LEFT_SENSOR);
38    Serial.print("    ");
39    Serial.print("RIghtsenor: ");
40    Serial.print(RIGHT_SENSOR);
41
42    if (RIGHT_SENSOR == 0 && LEFT_SENSOR == 0) {
43      forward(); //FORWARD
44      Serial.println("   forword");
45    }
46    else if (RIGHT_SENSOR == 1 && LEFT_SENSOR == 0) {
47      right(); //Move Right
48      Serial.println("   right turn");
49    }
50    else if (RIGHT_SENSOR == 0 && LEFT_SENSOR == 1) {
51      left(); //Move Left
52      Serial.println("   left turn");
```

Creating a Loop for IR sensors to continuingly providing feedback to the Arduino board for perceiving the lines and making decisions according to it.

Defining the If and Else Statement to execute the task if the left sensor detects the black line, move right or else move forward, and if the right sensor detects the black line, move left or else move forward.

line_follow | Arduino IDE 2.1.0

File Edit Sketch Tools Help

Arduino Uno

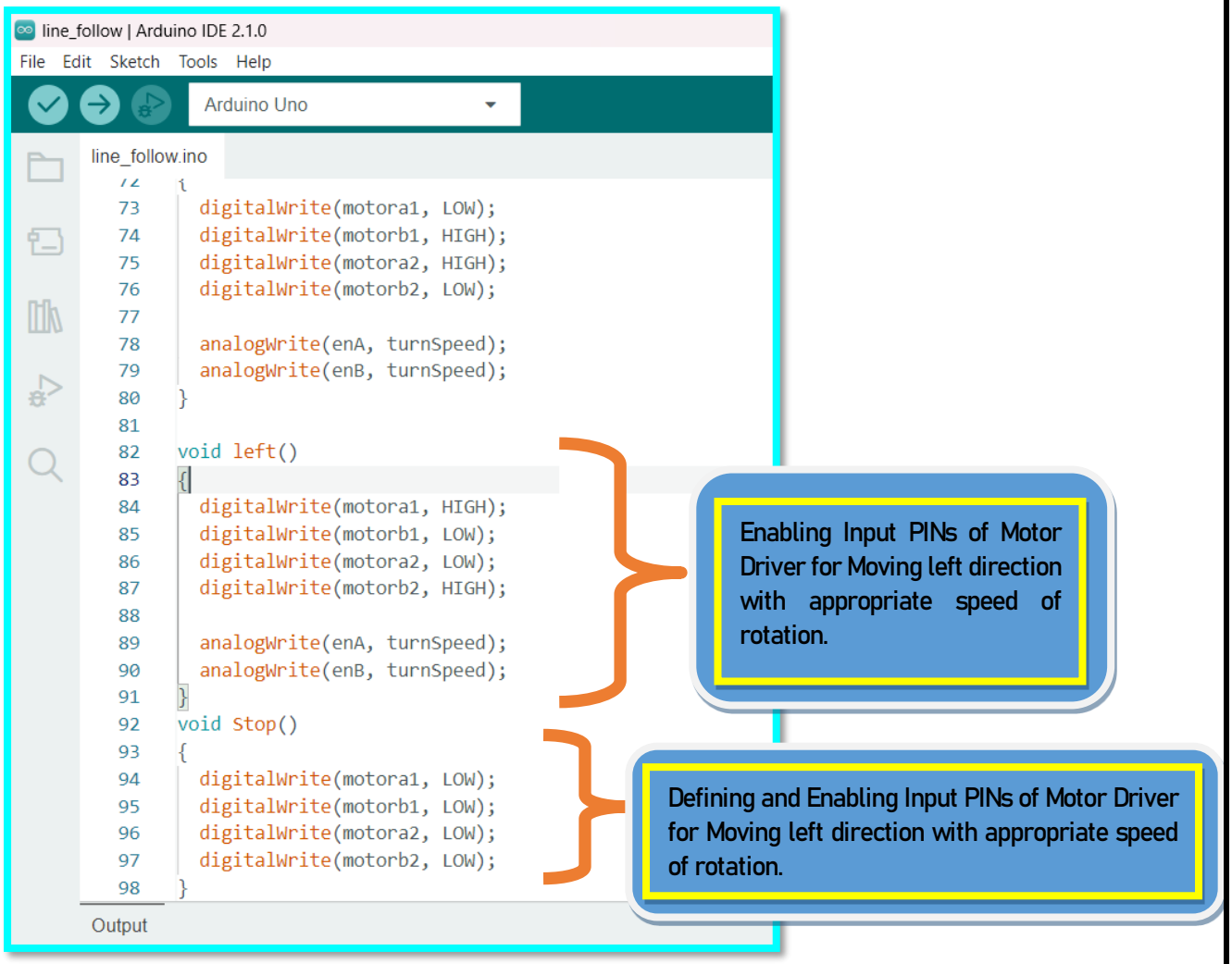
line_follow.ino

```
53    }
54    else if (RIGHT_SENSOR == 1 && LEFT_SENSOR == 1) {
55      Serial.println("Stop");
56      Stop();
57    }
58  }
59
60  void forward()
61  {
62    digitalWrite(motora1, HIGH);
63    digitalWrite(motorb1, LOW);
64    digitalWrite(motora2, HIGH);
65    digitalWrite(motorb2, LOW);
66
67    analogWrite(enA, maxSpeed);
68    analogWrite(enB, maxSpeed);
69  }
70
71  void right()
72  {
73    digitalWrite(motora1, LOW);
74    digitalWrite(motorb1, HIGH);
75    digitalWrite(motora2, HIGH);
76    digitalWrite(motorb2, LOW);
77
78    analogWrite(enA, turnSpeed);
```

If Both the Sensors detect black colour, then stop.

Enabling Input PINs of Motor Driver for Moving forward with appropriate speed of rotation.

Enabling Input PINs of Motor Driver for Moving right with appropriate speed of rotation.



There are four conditions in this line following robot that we read by using Arduino. We have used two sensors namely the left sensor and the right sensor.

Input		Output				Movement Of Robot
Left Sensor	Right Sensor	Left Motor		Right Motor		
LS	RS	LM1	LM2	RM1	RM2	
0	0	0	0	0	0	Stop
0	1	1	0	0	0	Turn Right
1	0	0	0	1	0	Turn Left
1	1	1	0	1	0	Forward

In today's technologically advanced society, embedded systems have become essential. These technologies enable major improvements in autonomous driving and related fields through effective line detection and intelligent road symbol classification. The line following robot project presents a fascinating chance to explore the world of robotics and autonomous systems. By fusing the fields of engineering, electronics, and programming. Extending this, We can dive into the World of Machine Learning and Deep Learning by Exploring Behavioural Cloning in Autonomous Systems and Intelligent Classification of Road Symbols.

Road Sense: Behaviour Cloning for Autonomous Driving Systems and Intelligent Road Symbol Classification

Classifying intelligent road symbols:

A crucial component of autonomous driving systems is the classification of intelligent road symbols. Advanced machine learning techniques are used by embedded systems to enable automobiles to comprehend and react appropriately to road signs and symbols. These algorithms successfully identify and categorise road symbols by utilising computer vision techniques and neural networks. Vehicles can make educated decisions based on the identified symbols by incorporating intelligent road symbol categorization into autonomous driving systems, assuring increased safety and effectiveness on the roads.

Behaviour Cloning for Autonomous Driving Systems:

Behaviour cloning is another fundamental concept in autonomous driving systems. By recording human driver behaviour and training an embedded system to mimic these actions, vehicles can learn from human expertise and navigate roads autonomously. This technique involves collecting data from various sensors, such as cameras, LiDAR, and GPS, and training the embedded system using machine learning algorithms. The system learns to map sensor inputs to appropriate control actions, enabling the vehicle to mimic human driving behaviour effectively. Two Explain these two. I have Gathered Various types of Images from all three angles of an autonomous car using Udacity Autonomous Car Simulator and For Classifying Road Symbols, we can use Perceptron Based and Convolutional Neural Networks based machine learning models, therefore to know it better let's dive into the depth of codes for executing the purpose of Classification of Road Symbol and Behavioural Cloning.

THE PERCEPTRON:

Perceptron.ipynb

File Edit View Insert Runtime Tools Help Last edited on July 6

+ Code + Text

```
import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
import matplotlib.pyplot as plt
%matplotlib inline
```

```
n_pts = 500
np.random.seed(0)
Xa = np.array([np.random.normal(13, 2, n_pts),
               np.random.normal(12, 2, n_pts)]).T
Xb = np.array([np.random.normal(8, 2, n_pts),
               np.random.normal(6, 2, n_pts)]).T

X = np.vstack((Xa, Xb))
y = np.matrix(np.append(np.zeros(n_pts), np.ones(n_pts))).T

plt.scatter(X[:n_pts,0], X[:n_pts,1])
plt.scatter(X[n_pts:,0], X[n_pts:,1])
```

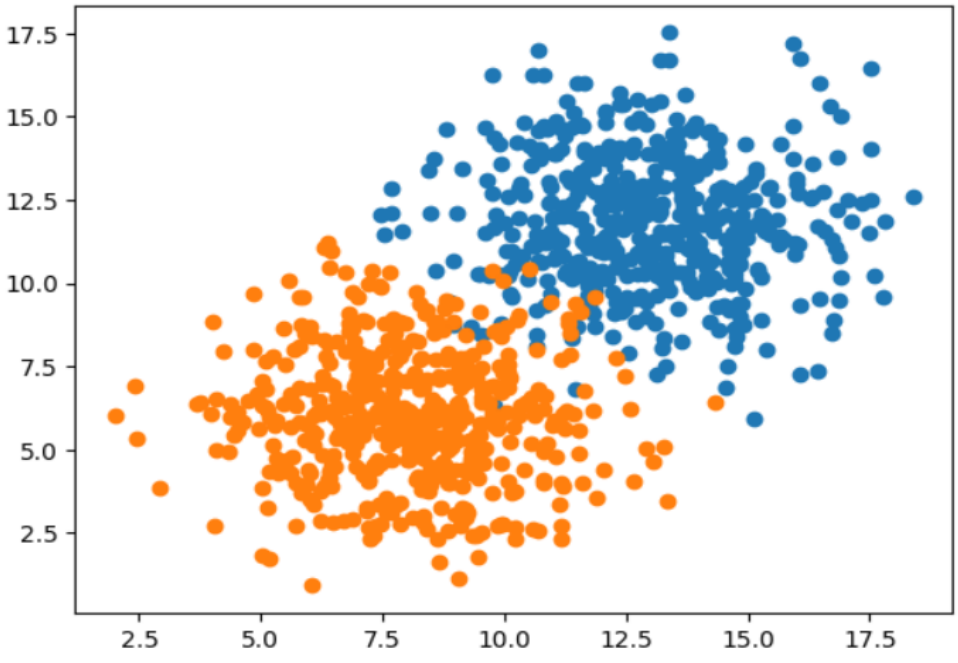
```
<matplotlib.collections.PathCollection at 0x7f2a26f9a410>
```

Perceptron.ipynb

File Edit View Insert Runtime Tools Help Last edited on July 6

+ Code + Text

```
<matplotlib.collections.PathCollection at 0x7f2a26f9a410>
```



Perceptron.ipynb

File Edit View Insert Runtime Tools Help Last edited on July 6

+ Code + Text

```
model = Sequential()
model.add(Dense(units = 1, input_shape =(2,), activation='sigmoid'))
adam = Adam(lr = 0.1)
model.compile(adam, loss ='binary_crossentropy', metrics = ['accuracy'])
h = model.fit(x=X, y=y, verbose=1, batch_size =50, epochs = 500, shuffle = 'true')
```

```
Epoch 472/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0709 - accuracy: 0.9710
Epoch 473/500
20/20 [==============================] - 0s 2ms/step - loss: 0.0653 - accuracy: 0.9730
Epoch 474/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0661 - accuracy: 0.9740
Epoch 475/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0800 - accuracy: 0.9680
Epoch 476/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0680 - accuracy: 0.9750
Epoch 477/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0639 - accuracy: 0.9760
Epoch 478/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0719 - accuracy: 0.9720
Epoch 479/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0683 - accuracy: 0.9790
Epoch 480/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0662 - accuracy: 0.9730
Epoch 481/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0663 - accuracy: 0.9790
Epoch 482/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0808 - accuracy: 0.9730
Epoch 483/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0920 - accuracy: 0.9690
Epoch 484/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0702 - accuracy: 0.9710
Epoch 485/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0634 - accuracy: 0.9770
```

Perceptron.ipynb

File Edit View Insert Runtime Tools Help Last edited on July 6

+ Code + Text

```
Epoch 485/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0634 - accuracy: 0.9770
Epoch 486/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0667 - accuracy: 0.9730
Epoch 487/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0738 - accuracy: 0.9760
Epoch 488/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0683 - accuracy: 0.9780
Epoch 489/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0673 - accuracy: 0.9750
Epoch 490/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0599 - accuracy: 0.9780
Epoch 491/500
20/20 [==============================] - 0s 2ms/step - loss: 0.0633 - accuracy: 0.9780
Epoch 492/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0628 - accuracy: 0.9770
Epoch 493/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0673 - accuracy: 0.9710
Epoch 494/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0605 - accuracy: 0.9780
Epoch 495/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0703 - accuracy: 0.9730
Epoch 496/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0663 - accuracy: 0.9720
Epoch 497/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0647 - accuracy: 0.9750
Epoch 498/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0620 - accuracy: 0.9760
Epoch 499/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0696 - accuracy: 0.9700
Epoch 500/500
20/20 [==============================] - 0s 1ms/step - loss: 0.0669 - accuracy: 0.9730
```

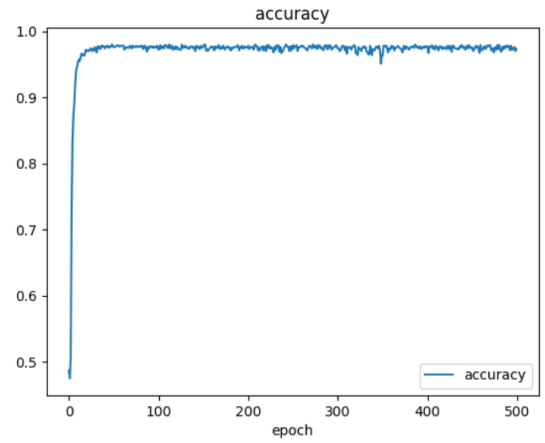
Perceptron.ipynb

File Edit View Insert Runtime Tools Help Last edited on July 6

+ Code + Text

```
plt.plot(h.history['accuracy'])
plt.title('accuracy')
plt.xlabel('epoch')
plt.legend(['accuracy'])
```

```
<matplotlib.legend.Legend at 0x7f2a22a27d30>
```



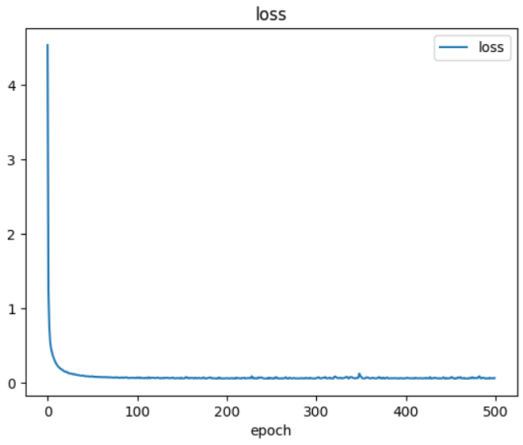
Perceptron.ipynb

File Edit View Insert Runtime Tools Help Last edited on July 6

+ Code + Text

```
plt.plot(h.history['loss'])
plt.title('loss')
plt.xlabel('epoch')
plt.legend(['loss'])
```

```
<matplotlib.legend.Legend at 0x7f2a244ecaf0>
```



Perceptron.ipynb

File Edit View Insert Runtime Tools Help Last edited on July 6

+ Code + Text

```
[ ] def plot_decision_boundary(X, y, model):
        x_span = np.linspace(min(X[:,0]) - 1, max(X[:,0]) + 1)
        y_span = np.linspace(min(X[:,1]) - 1, max(X[:,1]) + 1)
        xx, yy = np.meshgrid(x_span, y_span)
        xx_, yy_ = xx.ravel(), yy.ravel()
        grid = np.c_[xx_, yy_]
        pred_func = model.predict(grid)
        z = pred_func.reshape(xx.shape)
        plt.contourf(xx, yy, z)
```

```
[ ] plot_decision_boundary(X, y, model)
    plt.scatter(X[:n_pts,0], X[:n_pts,1])
    plt.scatter(X[n_pts:,0], X[n_pts:,1])

    plot_decision_boundary(X, y, model)
    plt.scatter(X[:n_pts,0], X[:n_pts,1])
    plt.scatter(X[n_pts:,0], X[n_pts:,1])
    x = 7.5
    y = 5


    point = np.array([[x, y]])
    prediction = model.predict(point)
    plt.plot([x], [y], marker='o', markersize=10, color="red")
    print("prediction is: ",prediction)
```

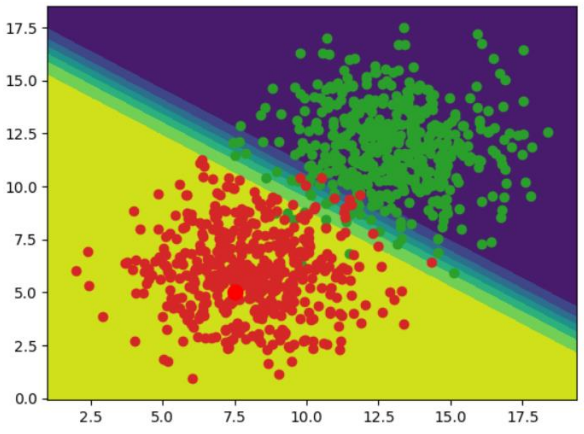
```
79/79 [==============================] - 0s 2ms/step
79/79 [==============================] - 0s 1ms/step
1/1 [==============================] - 0s 25ms/step
```

Perceptron.ipynb

File Edit View Insert Runtime Tools Help Last edited on July 6

+ Code + Text

```
79/79 [==============================] - 0s 2ms/step
79/79 [==============================] - 0s 1ms/step
1/1 [==============================] - 0s 25ms/step
prediction is:  [[0.99999595]]
```



The Perceptron is trained to receive inputs and the form of input nodes and transfer the appropriate output similar to how with neurons, the dendrite receives electrical signals while the axon branches. And are very useful in Supervised machine learning In other words, in supervised learning, a learner is provided examples of certain inputs and outputs, and based on that training, given a new input, it's able to predict the corresponding outputs. linear regression, a supervised learning algorithm that allows us to make predictions based on linearly arranged data sets. The Above code is an example of a Linear Regression model this is one of the fundamental building blocks for working on Multiclass classification and behavioural cloning techniques. The Linear Model consists of Sigmoid function and their implementation, Cross entropy and Gradient Descent also play a vital role in working on a perceptron the above code fully described the techniques.

Deep Neural Networks:

As we become experts in deep neural networks, we will eventually become experts in convolutional neural networks, which we'll employ in our scenario to extract particular features from photos, utilizing it to extract picture characteristics from traffic signs, assisting us in classifying and extracting these self-driving car simulations use picture data from a specific truck component to help forecast the proper angle for steering.

An artificial neural network (ANN) with the capability of learning and modelling complicated patterns and representations from data is known as a deep neural network (DNN). It draws inspiration from the way that neurons are connected in layers in the human brain, which processes and transmits information.

Multiple hidden layers between the input and output layers are referred regarded as "deep" layers in deep neural networks. Deep neural networks, which can develop hierarchical representations of data, can include tens or even hundreds of hidden layers in contrast to traditional neural networks, which may only have one or two.

A list of a deep neural network's essential parts:

Input Layer: The neural network's input layer is where it all begins. It accepts unprocessed data, such as pictures, text, or numerical features, and sends it to the top hidden layer for processing.

Hidden layer: Where the magic in a deep neural network happens are the hidden layers. There are numerous linked neurons (or nodes) in each buried layer. Each neuron in a layer receives input from the neurons in the layer below, processes it using a set of weights and biases, and then sends the processed information to the neurons in the layer above. Up until the data reaches the output layer, this process is repeated for each layer.

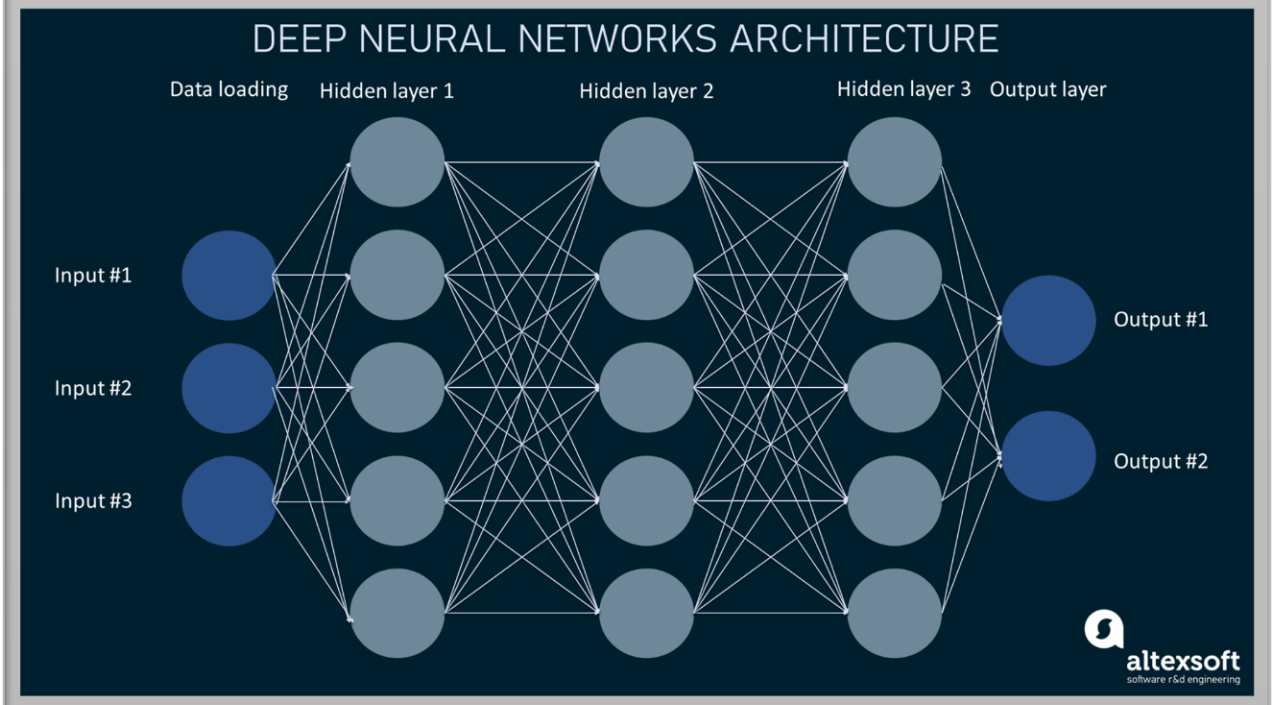
Neurons (Nodes): Each neuron in a neural network is a discrete computational unit. Each neuron performs a straightforward calculation: it adds a bias term to the weighted sum of its inputs and then applies an activation function to get an output. The network gains non-linearity from the activation function, allowing it to understand intricate relationships in the data.

Weights and Biases: A neural network's learnable parameters are weights and biases. To reduce the discrepancy between its anticipated outputs and the actual objectives in the training data, the network modifies these parameters during the training process. The biases enable neurons to change their activation thresholds, whereas the weights control the strength of connections between neurons.

Activation Function: Based on the information it receives, the activation function decides whether a neuron should be activated (i.e., fire). The Tanh, Sigmoid, Rectified Linear Unit (ReLU), and other activation mechanisms are frequently used. For the network to successfully approximate complicated functions, the activation function introduces non-linearities.

Output Layer: The output layer provides the ultimate predictions or outcomes of the neural network's calculation. The sort of task the neural network is intended for determines how many neurons are present in the output layer. When classifying images, for instance, the output layer might have neurons corresponding to many classes, with each neuron denoting the likelihood that the input image belongs to that class.

Deep neural networks are widely used in various fields, including computer vision, natural language processing, speech recognition, and many others. They have shown remarkable performance in tasks such as image and speech recognition, language translation, that's why they are very essential for making of this project during this project I worked upon making a deep neural network so that the data can be classified accurately and the task of behavioural cloning can be done effortlessly.



Deep_Neural_network.ipynb

File Edit View Insert Runtime Tools Help Saving...

+ Code + Text

```
[3] import numpy as np
    import matplotlib.pyplot as plt
    import keras
    from sklearn import datasets
    from keras.models import Sequential
    from keras.layers  import Dense
    from keras.optimizers import Adam
```

```
[4] np.random.seed(0)
```

```
n_pts = 500
X, y = datasets.make_circles(n_samples=n_pts, random_state = 123, noise=0.1, factor = 0.2)
print(X)
print(y)
```

```
[-8.81623073e-01  4.65409202e-01]
[-7.65765970e-02  8.70242992e-02]
[-8.35303067e-01 -1.49797356e-01]
[-3.46716314e-01 -1.07721046e+00]
[ 7.87432212e-02 -2.59708139e-01]
[ 2.19690108e-01 -1.55990310e-01]
[-6.00562059e-01  8.43427575e-01]
[ 2.92841842e-01  1.55886284e-02]
[ 6.74713131e-01  7.07239729e-01]
[-2.33192512e-01  1.45186279e-01]
[ 1.21530709e+00  1.17672930e-01]
[-4.00973653e-02 -1.96212152e-01]
[ 2.96095156e-02  1.42989320e-01]
[ 9.88037620e-02  1.80812836e-01]
[-1.67723363e-01  1.11648710e+00]
[ 3.21761251e-01 -9.66364289e-01]
```

+ Code + Text

```
[-1.30370035e-01 -2.69468536e-01]
[ 6.37286375e-01  8.75101351e-01]
[-1.43301315e-01  9.44116672e-01]
[ 3.97072425e-01  7.65633605e-01]
[-1.01761342e+00 -9.16862411e-03]
[-8.40296123e-01  3.29722054e-01]
[-7.02652140e-01  6.95788156e-01]
[ 8.35917113e-01 -7.27461170e-01]
[ 5.76183846e-01 -6.42799752e-01]
[ 2.20179935e-01  1.03988488e+00]
[-9.92065426e-01  1.16188458e-01]
[-9.82625390e-01  3.01573904e-01]
[-5.30935997e-01  9.23976060e-01]
[ 9.20888959e-01  5.15771449e-01]
[ 9.11451732e-01 -4.77269454e-01]
[-6.45713806e-01  7.40928939e-01]
[ 1.54897389e-01 -2.23758776e-01]
[-1.38035949e-01  1.78181233e-01]
[-2.33594005e-01  1.30771621e-01]
[-3.61730047e-01  4.33121987e-02]]
[0 1 1 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 1 1 0 0 1 0 1 0 1 1 0 0 0 1 0 1 0 1 1 0
 1 0 1 1 0 0 1 0 1 0 0 0 0 1 0 0 0 1 1 1 1 0 0 0 1 0 1 1 1 0 0 0 0 0 0 0 0 1
 1 0 0 0 1 0 1 1 1 0 1 1 0 1 1 0 0 0 0 0 0 1 0 1 1 1 0 1 0 0 0 0 1 1 1 0 1 1
 1 1 0 0 1 0 1 1 0 0 1 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 1 0 0 1 0 0 1 0 1 0 1 1
 1 1 0 0 0 1 0 0 1 0 1 1 1 1 0 0 1 1 0 0 0 1 0 1 1 0 1 1 1 1 1 0 1 0 1 1 0 0
 1 0 1 0 1 1 0 0 1 0 0 1 0 1 1 1 1 0 0 1 0 0 0 0 0 1 0 1 1 0 0 1 1 0 0 0 0 0
 0 0 0 0 1 1 0 1 0 1 0 1 0 1 1 1 1 1 1 1 0 1 0 0 0 0 1 0 1 0 1 0 0 1 0 1 0 0 0
 0 1 1 1 0 0 1 0 0 0 1 1 1 0 0 1 0 0 1 0 0 1 1 1 1 1 1 0 1 1 1 0 0 0 1 0 0 0
 0 0 0 1 0 0 1 1 1 0 1 1 0 0 1 0 1 0 1 0 1 1 0 0 0 0 0 0 1 1 1 0 1 0
 0 1 1 1 1 1 0 1 0 1 1 0 0 0 0 0 1 0 1 1 0 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 0
 0 0 1 0 0 1 0 1 0 1 1 1 1 1 0 0 1 0 0 1 0 1 1 1 0 0 1 0 1 0 1 1 1 1 1 1 0 1 1
 0 0 1 1 1 0 0 1 1 0 1 1 0 1 0 0 1 1 0 1 0 1 0 1 1 1 0 0 1 1 1 1 1 1 0 0 1
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1]
```

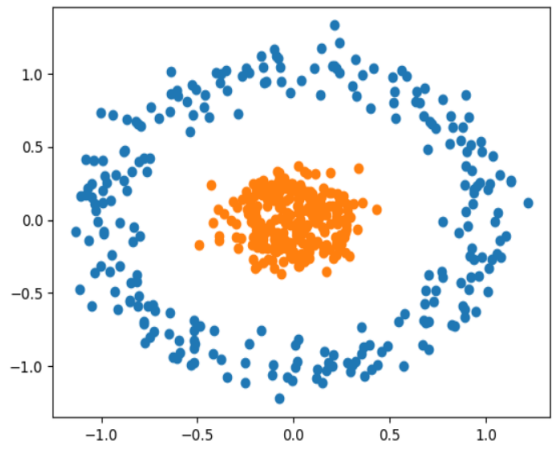
Deep_Neural_network.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
plt.scatter(X[y==0, 0], X[y==0,1])
plt.scatter(X[y==1, 0], X[y==1,1])
```

```
<matplotlib.collections.PathCollection at 0x7ddfab4d9f60>
```



Deep_Neural_network.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
[7] model = Sequential()
    model.add(Dense(4, input_shape=(2,), activation='sigmoid'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(Adam(lr = 0.01), 'binary_crossentropy', metrics=['accuracy'])
```

```
/usr/local/lib/python3.10/dist-packages/keras/optimizers/legacy/adam.py:117: UserWarning:
  super().__init__(name, **kwargs)
```

```
[8] history = model.fit(x=X, y=y , verbose =1 , batch_size = 20, epochs =100 , shuffle='true')
```

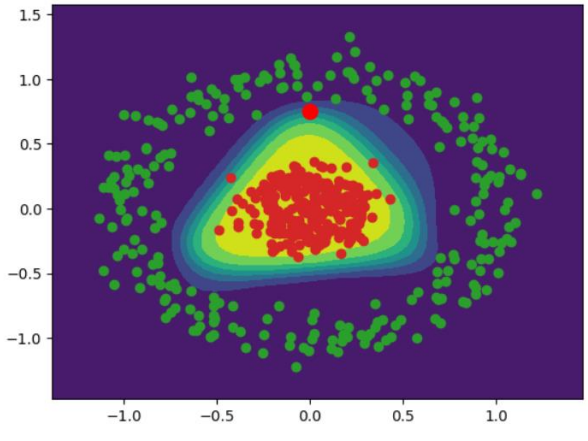
```
Epoch 72/100
25/25 [==============================] - 0s 2ms/step - loss: 0.0618 - accuracy: 0.9980
Epoch 73/100
25/25 [==============================] - 0s 2ms/step - loss: 0.0601 - accuracy: 0.9980
Epoch 74/100
25/25 [==============================] - 0s 2ms/step - loss: 0.0585 - accuracy: 0.9980
Epoch 75/100
25/25 [==============================] - 0s 1ms/step - loss: 0.0570 - accuracy: 0.9980
Epoch 76/100
25/25 [==============================] - 0s 1ms/step - loss: 0.0555 - accuracy: 0.9980
Epoch 77/100
25/25 [==============================] - 0s 1ms/step - loss: 0.0540 - accuracy: 0.9980
Epoch 78/100
25/25 [==============================] - 0s 1ms/step - loss: 0.0527 - accuracy: 0.9980
Epoch 79/100
25/25 [==============================] - 0s 2ms/step - loss: 0.0514 - accuracy: 0.9980
Epoch 80/100
25/25 [==============================] - 0s 1ms/step - loss: 0.0501 - accuracy: 0.9980
Epoch 81/100
25/25 [==============================] - 0s 1ms/step - loss: 0.0489 - accuracy: 0.9980
Epoch 82/100
25/25 [==============================] - 0s 1ms/step - loss: 0.0477 - accuracy: 0.9980
Epoch 83/100
25/25 [==============================] - 0s 1ms/step - loss: 0.0466 - accuracy: 0.9980
```

```
def plot_decision_boundary(X, y, model):
    x_span = np.linspace(min(X[:,0]) - 0.25, max(X[:,0]) + 0.25)
    y_span = np.linspace(min(X[:,1]) - 0.25, max(X[:,1]) + 0.25)
    xx, yy = np.meshgrid(x_span, y_span)
    grid = np.c_[xx.ravel(), yy.ravel()]
    pred_func = model.predict(grid)
    z = pred_func.reshape(xx.shape)
    plt.contourf(xx, yy, z)
plot_decision_boundary(X, y, model)
plt.scatter(X[y==0, 0], X[y==0, 1])
plt.scatter(X[y==1, 0], X[y==1, 1])
plot_decision_boundary(X, y, model)
plt.scatter(X[y==0, 0], X[y==0, 1])
plt.scatter(X[y==1, 0], X[y==1, 1])

x = 0
y = 0.75

point = np.array([[x, y]])
predict = model.predict(point)
plt.plot([x], [y], marker='o', markersize=10, color="red")
print("Prediction is: ", predict)
```

```
79/79 [==============================] - 0s 798us/step
79/79 [==============================] - 0s 836us/step
1/1 [==============================] - 0s 18ms/step
Prediction is:  [[0.41325173]]
```

```
plt.plot([x], [y], marker='o', markersize=10, color="red")
print("Prediction is: ", predict)
```

```
79/79 [==============================] - 0s 798us/step
79/79 [==============================] - 0s 836us/step
1/1 [==============================] - 0s 18ms/step
Prediction is:  [[0.41325173]]
```



Multi-Class Classification:

Then Further to fulfil the requirements of my project I worked over Multiclass Classification of Data. Because in the context of classifying road symbols, multiclass classification is crucial because roads are typically adorned with a diverse range of symbols, signs, and markings that serve different purposes. Some common road symbols include stop signs, yield signs, speed limit signs, pedestrian crossing symbols, lane markings, and many others. Being able to automatically and accurately identify these symbols is vital for various applications, including:

Road Safety:

Accurately identifying and categorizing road signs helps to improve general safety. Examples of warning signs that help drivers make better judgments include curves ahead, slick roads, and pedestrian crossings.

Autonomous Vehicles:

Computer vision systems are used by self-driving cars and other autonomous vehicles to decipher road signs and symbols. Autonomous vehicles must correctly recognize road symbols in order to operate safely and follow traffic laws.

Traffic Management:

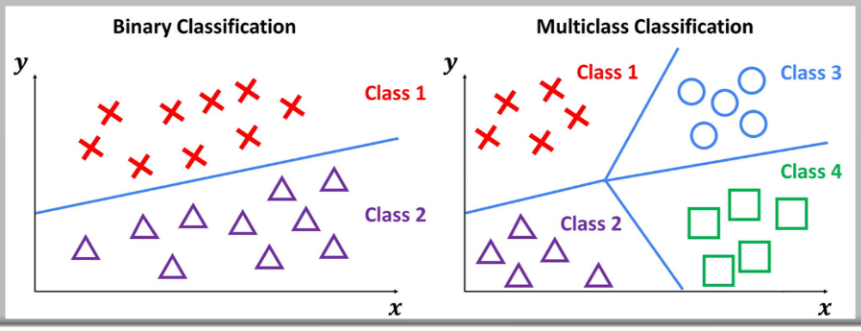
Assessing Road conditions, scheduling road maintenance, and maximizing traffic flow can all be done with the help of monitoring and analysing road symbols.

Understanding the distribution of road symbols around a city can help urban planners make informed decisions about how to improve infrastructure and raise pedestrian and driver safety.

To perform multiclass classification for road symbols, machine learning algorithms, particularly deep learning models like convolutional neural networks (CNNs), are commonly used. CNNs excel at image classification tasks, making them suitable for identifying and categorizing road symbols in images taken from cameras or other sensors.

Training a multiclass classification model for road symbols involves providing labelled datasets where each image is associated with the correct road symbol class. The model learns from this labelled data during the training process and adjusts its internal parameters (weights and biases) to make accurate predictions during inference on unseen data.

Overall, multiclass classification is crucial for precisely and effectively categorizing road symbols, which will improve traffic management, road safety, and driving experiences in the era of smart transportation systems.



Multiclass.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
import numpy as np
import keras
from sklearn import datasets
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
from keras.utils.np_utils import to_categorical
```

```
n_pts = 500
centers = [[-1,1], [-1,-1],[1,-1]]
X, y = datasets.make_blobs(n_samples=n_pts, random_state=123, centers=centers,cluster_std=0.4)
print(X)
print(y)
```

```
[[ 0.95219249 -0.60923137]
 [ 1.65952714 -0.31381256]
 [ 0.526137   -0.94160831]
 [-1.41959667  0.96995176]
 [-0.94002647 -0.78014296]
 [-1.60724319  0.81552503]
 [-0.82564969  0.8724172 ]
 [ 0.88319833 -1.0467727 ]
 [ 1.72626083 -1.10110012]
 [ 1.42796744 -1.36373081]
 [-0.91416557 -1.16086389]
 [-0.61034703 -0.93588251]
 [-1.58604649  1.14537822]
 [-0.44909701  0.94272961]
```

Multiclass.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
[4]
[ 0.98079432 -1.44501172]
[ 0.92110545 -1.81683857]
[-0.8709693   0.82633339]
[ 0.95153328 -1.80343419]
[ 1.4290519  -1.32931407]
[-0.85139013 -1.06344208]
[-1.1340043   1.41244578]
[-1.30345137 -1.48001731]
[ 0.86351757 -1.19591191]
[-0.96261567  1.58383571]
[ 0.43488821 -0.80519628]
[-1.76948629  1.27831493]
[ 1.080264   -1.34128061]
[-1.10617825 -0.62816621]
[-1.03808102  1.11147341]
[ 1.34998493 -0.85909137]
[-0.83876562  1.58877175]
[-1.11660269 -0.90184824]
[-0.63827751 -0.52000494]
[-1.32706732  0.46145698]]
[2 2 2 0 1 0 0 2 2 2 1 1 0 0 1 2 2 2 2 0 2 0 1 2 1 2 0 0 2 1 1 0 0 2 1 0 2
 0 2 0 0 2 1 0 2 0 1 2 0 2 0 1 1 0 1 2 2 0 1 2 0 2 0 2 1 0 1 1 1 1 2 1 1 1
 1 1 2 1 1 2 0 1 2 2 1 0 2 1 1 2 0 0 2 1 2 2 0 1 2 0 2 1 1 2 2 0 0 1 2 1 1
 2 0 0 1 0 2 2 2 2 1 2 1 2 0 2 2 0 1 0 0 0 0 2 0 1 0 2 2 0 2 2 2 2 1 2 0 0
 0 2 1 0 2 0 0 1 1 0 2 0 0 1 1 2 2 0 2 1 2 2 2 1 1 1 2 0 0 1 0 1 2 2 1 1 2
 1 1 0 0 2 2 1 0 0 0 0 2 2 0 2 1 1 2 0 0 2 2 0 1 0 2 0 2 2 0 0 2 1 2 0 2 0
 1 1 2 0 0 0 2 1 2 2 2 0 0 1 0 2 0 2 0 2 1 1 1 2 2 1 1 1 0 1 2 1 0 1 2 2 0
 0 0 1 1 1 2 1 1 2 1 1 2 0 0 0 2 0 2 0 0 0 2 1 1 0 2 1 0 0 1 1 1 1 2 1 1 0
 0 0 2 1 1 1 0 0 0 0 0 0 0 1 2 1 2 1 0 2 0 0 1 2 0 1 2 2 2 2 2 0 1 2 0 2 1
 0 0 1 2 0 2 0 1 1 0 2 1 0 1 1 1 0 2 0 2 0 1 1 0 1 1 1 2 2 0 2 0 2 2 1 1 0
 2 2 2 1 1 1 2 2 0 1 2 1 2 0 1 2 2 2 0 1 1 1 2 0 2 0 0 2 1 1 1 2 0 1 0 0 1
 1 2 0 0 1 2 1 1 0 2 0 1 0 2 0 0 0 1 1 1 2 0 1 1 0 2 1 2 1 1 1 1 0 0 0 0 0
 2 2 1 1 0 1 0 2 2 2 2 0 0 1 1 0 0 1 1 1 2 2 1 1 1 1 0 0 2 0 1 0 1 0 1 1 2
 2 0 2 2 1 0 1 2 0 2 0 2 1 0 2 0 1 1 0]
```

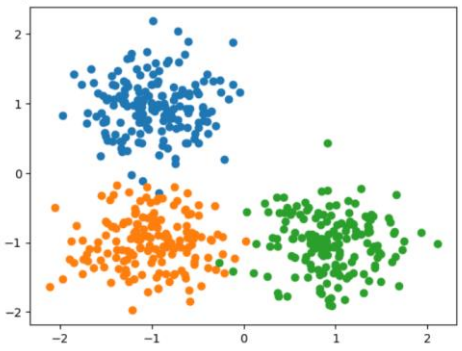
Multiclass.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
plt.scatter(X[y==0, 0], X[y==0, 1])
plt.scatter(X[y==1, 0], X[y==1, 1])
plt.scatter(X[y==2, 0], X[y==2, 1])
plt.scatter(X[y==3, 0], X[y==3, 1])
plt.scatter(X[y==4, 0], X[y==4, 1])
```

```
<matplotlib.collections.PathCollection at 0x7ac5eea735e0>
```



Multiclass.ipynb

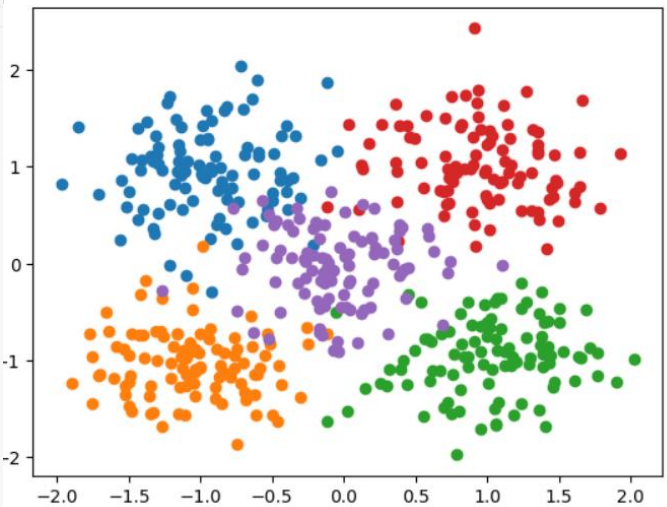
File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
history = model.fit(X, y_cat, verbose=1, batch_size = 50, epochs=100)
```

```
Epoch 72/100
10/10 [==============================] - 0s 7ms/step - loss: 0.0273 - accuracy: 0.9900
Epoch 73/100
10/10 [==============================] - 0s 7ms/step - loss: 0.0277 - accuracy: 0.9900
Epoch 74/100
10/10 [==============================] - 0s 6ms/step - loss: 0.0281 - accuracy: 0.9900
Epoch 75/100
10/10 [==============================] - 0s 7ms/step - loss: 0.0267 - accuracy: 0.9900
Epoch 76/100
10/10 [==============================] - 0s 8ms/step - loss: 0.0275 - accuracy: 0.9900
Epoch 77/100
10/10 [==============================] - 0s 15ms/step - loss: 0.0266 - accuracy: 0.9900
Epoch 78/100
10/10 [==============================] - 0s 11ms/step - loss: 0.0267 - accuracy: 0.9900
Epoch 79/100
10/10 [==============================] - 0s 7ms/step - loss: 0.0268 - accuracy: 0.9900
Epoch 80/100
10/10 [==============================] - 0s 8ms/step - loss: 0.0267 - accuracy: 0.9900
Epoch 81/100
10/10 [==============================] - 0s 3ms/step - loss: 0.0271 - accuracy: 0.9900
Epoch 82/100
10/10 [==============================] - 0s 7ms/step - loss: 0.0275 - accuracy: 0.9900
Epoch 83/100
10/10 [==============================] - 0s 4ms/step - loss: 0.0263 - accuracy: 0.9900
Epoch 84/100
10/10 [==============================] - 0s 4ms/step - loss: 0.0260 - accuracy: 0.9900
Epoch 85/100
10/10 [==============================] - 0s 6ms/step - loss: 0.0262 - accuracy: 0.9900
Epoch 86/100
10/10 [==============================] - 0s 7ms/step - loss: 0.0263 - accuracy: 0.9900
Epoch 87/100
10/10 [==============================] - 0s 10ms/step - loss: 0.0259 - accuracy: 0.9900
Epoch 88/100
```

```
def plot_multiclass_decision_boundary(X, y, model):
    x_span = np.linspace(min(X[:,0]) - 1, max(X[:,0]) + 1)
    y_span = np.linspace(min(X[:,1]) - 1, max(X[:,1]) + 1)
    xx, yy = np.meshgrid(x_span, y_span)
    grid = np.c_[xx.ravel(), yy.ravel()]
    predict_function = model.predict_classes(grid)
    z = predict_function.reshape(xx.shape)
    plt.contourf(xx, yy, z)
plot_multiclass_decision_boundary(X, y_cat, model)
plt.scatter(X[y==0, 0], X[y==0, 1])
plt.scatter(X[y==1, 0], X[y==1, 1])
plt.scatter(X[y==2, 0], X[y==2, 1])
plt.scatter(X[y==3, 0], X[y==3, 1])
plt.scatter(X[y==4, 0], X[y==4, 1])
plot_multiclass_decision_boundary(X, y_cat, model)
plt.scatter(X[y==0, 0], X[y==0, 1])
plt.scatter(X[y==1, 0], X[y==1, 1])
plt.scatter(X[y==2, 0], X[y==2, 1])
plt.scatter(X[y==3, 0], X[y==3, 1])
plt.scatter(X[y==4, 0], X[y==4, 1])

x = -0.5
y = -0.5

point = np.array([[x, y]])
prediction = model.predict_classes(point)
plt.plot([x], [y], marker='o', markersize=10, color="yellow")
print("Prediction is: ", prediction)
```



Then to classifying Road Symbols and behavioural cloning for autonomous systems we need to work upon MNIST image Recognition. Let's Dive into that and see the code of the same.

MNIST IMAGE RECOGNITION :

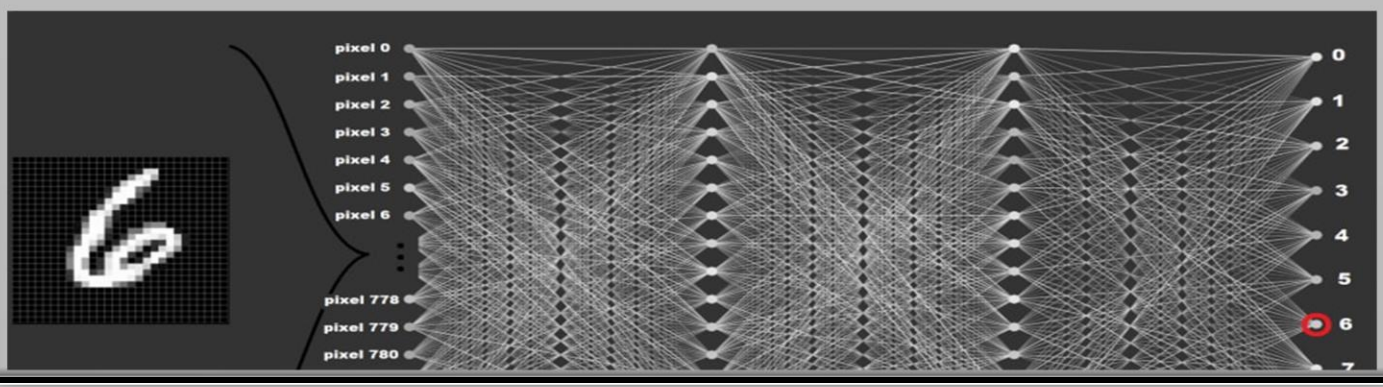
MNIST image recognition, which involves classifying handwritten digits, is not directly applicable to classifying road symbols or behavioural cloning in the context of autonomous vehicles. MNIST is a popular dataset used for training and evaluating machine learning models, particularly for image classification tasks. While the techniques used in MNIST classification can be applied to other image classification problems, there are significant differences between MNIST and road symbol recognition or behavioural cloning tasks.

Road Symbol Classification: Road symbols, such as traffic signs, traffic signals, or lane markings, have different characteristics and complexities compared to handwritten digits. Road symbols can have various shapes, colours, textures, and sizes. Additionally, they often contain more intricate details and exhibit greater variation across classes. Training a model on the MNIST dataset, which consists of simple grayscale handwritten digits, may not adequately capture the complexity and diversity of road symbols.

Behavioural Cloning: Behavioural cloning in the context of autonomous vehicles involves training a model to imitate human driving behaviour based on demonstrations. This task requires mapping visual input, such as camera images or video frames, to corresponding driving actions (e.g., steering angle, throttle, brake). While MNIST focuses on classifying static images, behavioural cloning requires processing dynamic visual data and associating it with continuous driving actions, which is fundamentally different from the MNIST task.

While the underlying principles of training deep learning models remain the same across various image classification tasks, including MNIST, the specific datasets, network architectures, and training procedures need to be tailored to the problem domain. For road symbol classification and behavioural cloning in autonomous vehicles, dedicated datasets and approaches that specifically address the characteristics and requirements of these tasks are more appropriate.

It's worth noting that there are specialized datasets available for road symbol recognition, such as the German Traffic Sign Recognition Benchmark (GTSRB) or the LISA Traffic Sign Dataset. These datasets contain real-world road symbol images captured under different conditions, which provide a more relevant and realistic training environment for developing road symbol classification models. Similarly, for behavioural cloning, datasets collected from driving simulators or real-world driving scenarios are utilized to train models that can effectively mimic human driving behaviour.



MNIST_image_Recognition.ipynb

File Edit View Insert Runtime Tools Help Last edited on July 12

+ Code + Text

```
import numpy as np
import matplotlib.pyplot as plt
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
from keras.utils.np_utils import to_categorical
import random
```

```
np.random.seed(0)
```

```
(X_train, y_train) , (X_test, y_test) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
```

```
print(X_train.shape)
print(X_test.shape)
print(y_train.shape[0])
```

```
(60000, 28, 28)
(10000, 28, 28)
60000
```

MNIST_image_Recognition.ipynb

File Edit View Insert Runtime Tools Help Last edited on July 12

+ Code + Text

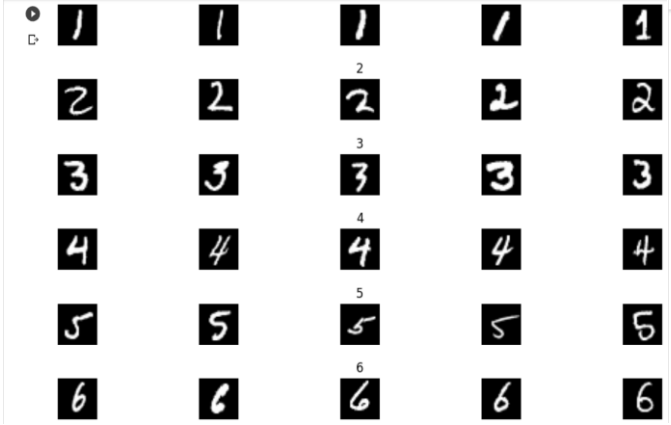
```
assert(X_train.shape[0]== y_train.shape[0]), "The number of images is not equal to the number of labels."
assert(X_test.shape[0]== y_test.shape[0]), "The number of images is not equal to the number of labels."
assert(X_train.shape[1:]== (28,28)), "The dimensions of the images are not 28x28."
assert(X_test.shape[1:]== (28,28)), "The dimensions of the images are not 28x28."
```

```
num_of_samples = []

cols = 5
num_classes = 10
fig, axs = plt.subplots(nrows=num_classes, ncols = cols, figsize=(10,10))
fig.tight_layout()
for i in range(cols):
  for j in range(num_classes):
    x_selected = X_train[y_train == j]
    axs[j][i].imshow(x_selected[random.randint(0,len(x_selected - 1)), :, :] ,cmap=plt.get_cmap("gray"))
    axs[j][i].axis("off")
    if i == 2:
      axs[j][i].set_title(str(j))
      num_of_samples.append(len(x_selected))
```

MNIST_image_Recognition.ipynb

File Edit View Insert Runtime Tools Help Last edited on July 12

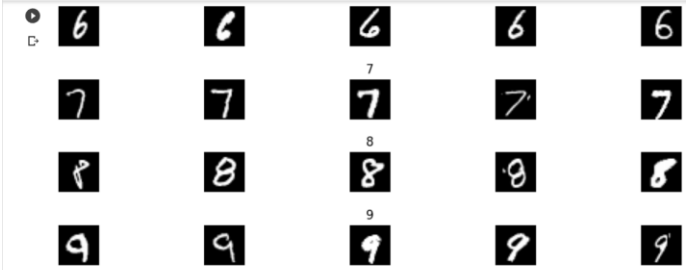
+ Code + Text



MNIST_image_Recognition.ipynb

File Edit View Insert Runtime Tools Help Last edited on July 12

+ Code + Text




```
print(num_of_samples)
plt.figure(figsize=(12,4))
plt.bar(range(0, num_classes), num_of_samples)
plt.title("Distribution of the training dataset")
plt.xlabel("Class number")
plt.ylabel("Number of images")
```

```
Text(0, 0.5, 'Number of images')[5923, 6742, 5958, 6131, 5842, 5421, 5918, 6265, 5851, 5949]
```

MNIST_image_Recognition.ipynb

File Edit View Insert Runtime Tools Help Last edited on July 12

+ Code + Text


```
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

```
X_train = X_train/255
X_test = X_test/255
```

MNIST_image_Recognition.ipynb

File Edit View Insert Runtime Tools Help Last edited on July 12

+ Code + Text

```
num_pixels = 784
X_train = X_train.reshape(X_train.shape[0], num_pixels)
X_test = X_test.reshape(X_test.shape[0], num_pixels)
print(X_test.shape)
```

```
(10000, 784)
```

```
def create_model():
  model = Sequential()
  model.add(Dense(10, input_dim=num_pixels, activation='relu'))
  model.add(Dense(10, activation='relu'))
  model.add(Dense(num_classes, activation='softmax'))
  model.compile(Adam(lr=0.01), loss='categorical_crossentropy', metrics=['accuracy'])
  return model
```

```
model = create_model()
print(model.summary())
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 10)                7850
=================================================================
```

```
model = create_model()
print(model.summary())
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 10)                7850

 dense_1 (Dense)             (None, 10)                110

 dense_2 (Dense)             (None, 10)                110

=================================================================
Total params: 8,070
Trainable params: 8,070
Non-trainable params: 0
_________________________________________________________________
None
/usr/local/lib/python3.10/dist-packages/keras/optimizers/legacy/adam.p
  super().__init__(name, **kwargs)
```

```
history = model.fit(X_train, y_train, validation_split=0.1, epochs =10, batch_size=200, verbose =1, shuffle =1)
```

```
Epoch 1/10
270/270 [==============================] - 2s 5ms/step - loss: 0.5384 - accuracy: 0.8363 - val_loss: 0.2376 - val_accuracy: 0.9307
Epoch 2/10
270/270 [==============================] - 1s 4ms/step - loss: 0.2845 - accuracy: 0.9177 - val_loss: 0.2264 - val_accuracy: 0.9400
Epoch 3/10
270/270 [==============================] - 1s 4ms/step - loss: 0.2500 - accuracy: 0.9288 - val_loss: 0.2065 - val_accuracy: 0.9397
Epoch 4/10
270/270 [==============================] - 1s 4ms/step - loss: 0.2324 - accuracy: 0.9333 - val_loss: 0.2043 - val_accuracy: 0.9428
Epoch 5/10
270/270 [==============================] - 1s 4ms/step - loss: 0.2236 - accuracy: 0.9347 - val_loss: 0.2002 - val_accuracy: 0.9457
Epoch 6/10
270/270 [==============================] - 2s 6ms/step - loss: 0.2177 - accuracy: 0.9367 - val_loss: 0.1971 - val_accuracy: 0.9458
Epoch 7/10
270/270 [==============================] - 2s 6ms/step - loss: 0.2119 - accuracy: 0.9384 - val_loss: 0.1984 - val_accuracy: 0.9477
Epoch 8/10
270/270 [==============================] - 1s 4ms/step - loss: 0.2082 - accuracy: 0.9389 - val_loss: 0.1925 - val_accuracy: 0.9480
Epoch 9/10
270/270 [==============================] - 1s 4ms/step - loss: 0.2012 - accuracy: 0.9417 - val_loss: 0.2283 - val_accuracy: 0.9385
Epoch 10/10
270/270 [==============================] - 1s 4ms/step - loss: 0.2003 - accuracy: 0.9425 - val_loss: 0.1943 - val_accuracy: 0.9438
```

MNIST_image_Recognition.ipynb

File Edit View Insert Runtime Tools Help Last edited on July 12

+ Code + Text

```
[ ] score = model.evaluate(X_test, y_test, verbose=0)
    print(type(score))
    print('Test score:', score[0])
    print('Test accuracy:', score[1])
```

```
<class 'list'>
Test score: 0.2219332456588745
Test accuracy: 0.9366000294685364
```

```
import requests
url = 'https://colah.github.io/posts/2014-10-Visualizing-MNIST/img/mnist_pca/MNIST-p1815-4.png'
response = requests.get(url, stream=True)
print(response)
```

```
<Response [200]>
```

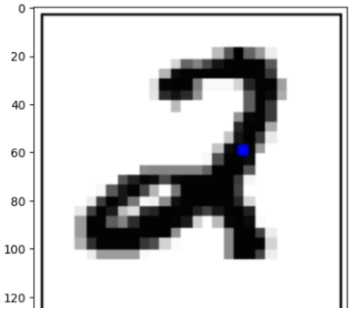
MNIST_image_Recognition.ipynb

File Edit View Insert Runtime Tools Help Last edited on July 12

+ Code + Text

```
import requests
from PIL import Image

url = 'https://colah.github.io/posts/2014-10-Visualizing-MNIST/img/mnist_pca/MNIST-p1815-4.png'
response = requests.get(url, stream=True)
img = Image.open(response.raw)
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x7f86b1350e50>
```




```
import cv2

img_array = np.asarray(img)
resized = cv2.resize(img_array, (28,28))
gray_scale = cv2.cvtColor(resized, cv2.COLOR_BGR2GRAY)
image = cv2.bitwise_not(gray_scale)
plt.imshow(image, cmap=plt.get_cmap("gray"))
print(gray_scale)
```

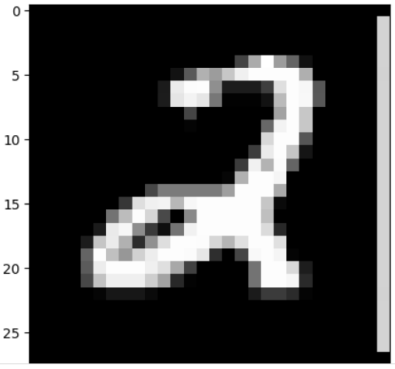
```
[[255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
  255 255 255 255 255 255 255 255 255 255]
 [255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
  255 255 255 255 255 255 255 255 255  45]
 [255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
  255 255 255 255 255 255 255 255 255  45]
 [255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
  255 255 255 255 255 255 255 255 255  45]
 [255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 188  85
    1 100 155 238 255 255 255 255 255  45]
 [255 255 255 255 255 255 255 255 255 255 255 255 237 166  79 100  57  16   3
    3   3   3  81 255 255 255 255 255  46]
 [255 255 255 255 255 255 255 255 255 255 228  21   4   3 113 227 227 227
  193  33   3   9 168 255 255 255 255  45]
 [255 255 255 255 255 255 255 255 255 255 227  27  12  26 136 253 253 253
  237  70   3   8 167 255 255 255 255  45]
 [255 255 255 255 255 255 255 255 255 255 255 255 186 255 255 255 255 255
  255  94   3  57 255 255 255 255 255  45]
 [255 255 255 255 255 255 255 255 255 255 255 255 251 255 255 255 255 255
  151  13   3  57 255 255 255 255 255  45]
 [255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
   40   3   3 169 255 255 255 255 255  45]
```

MNIST_image_Recognition.ipynb

File Edit View Insert Runtime Tools Help Last edited on July 12

+ Code + Text

```
[255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
 255 255 255 255 255 255 255 255 255 255  45]
[255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
 255 255 255 255 255 255 255 255 255 255  45]
[255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
 255 255 255 255 255 255 255 255 255 255  45]
[255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
 255 255 255 255 255 255 255 255 255 255  45]
[255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
 255 255 255 255 255 255 255 255 255 255 255]]
```



The above Predicted image by the code is number 2, This image Recognition helps in classifying Road symbols and plays a crucial role in Making of a Convolutional network of Classifying Road symbols and Behavioural Cloning. And Predicted a score and accuracy of the test of predicting a number.

After MNIST Image Recognition I worked on Building a Convolutional Neural Networks because Convolutional Neural Networks (CNNs) are highly important and widely used for classifying Road Symbols and behavioural cloning in the context of autonomous vehicles and advanced driver assistance systems (ADAS). Here's how CNNs play a crucial role in both tasks:

CONVOLUTIONAL NEURAL NETWORK:

Classifying Road Symbols:

Image Processing: Road symbols, such as traffic signs and markings, are visual elements. CNNs excel at image processing tasks and can effectively extract relevant features from images, making them well-suited for recognizing and classifying road symbols.

Hierarchical Feature Learning: CNNs can learn hierarchical representations of images, automatically discovering meaningful patterns and features at different levels of abstraction. This ability is particularly useful for classifying complex and diverse road symbols that may have varying shapes, colours, and orientations.

Robustness to Variations: Road symbols may appear under different lighting conditions, weather conditions, and viewpoints. CNNs are known for their robustness to such variations, ensuring reliable recognition of road symbols in real-world scenarios.

Behavioural Cloning:

Training Data from Demonstration: Behavioural cloning is a technique where an autonomous vehicle learns to imitate human drivers' behaviour by training on their driving demonstrations. CNNs can process visual input, such as images or video frames, and effectively map them to corresponding driving actions, enabling the vehicle to replicate the observed driving behaviour.

End-to-End Learning: CNNs allow for an end-to-end learning approach, where the raw visual input (e.g., camera feed) is directly fed into the neural network, and the network learns to output the desired driving actions (e.g., steering angle, throttle, brake) without the need for manual feature engineering.

Scalability: CNNs can handle large amounts of driving data, making it feasible to train models on extensive datasets collected from diverse driving scenarios, enabling the autonomous vehicle to learn from a wide range of experiences.

In both cases, CNNs have proven to be highly effective and efficient in handling complex visual data, making them a popular choice for computer vision tasks in the field of autonomous driving. They have significantly contributed to advancements in autonomous vehicles, providing solutions for object detection, lane detection, traffic sign recognition, and behavioural cloning tasks.

However, it's important to note that for safety-critical applications like autonomous driving, the deployment of CNN-based models requires rigorous testing, validation, and considerations for potential edge cases and safety measures to ensure the system's reliability and robustness.

Convolutional Neural Networks.ipynb ☆

File Edit View Insert Runtime Tools Help Last edited on July 13

+ Code + Text

```
import numpy as np
import matplotlib.pyplot as plt
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
from keras.utils.np_utils import to_categorical
from keras.models import Model
from keras.layers import Flatten
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.layers import Dropout
from keras.models import Model
import random


np.random.seed(0)




(X_train, y_train), (X_test, y_test)= mnist.load_data()

print(X_train.shape)
print(X_test.shape)
assert(X_train.shape[0] == y_train.shape[0]), "The number of images is not equal to the number of labels
assert(X_train.shape[1:] == (28,28)), "The dimensions of the images are not 28 x 28."
```

Convolutional Neural Networks.ipynb ☆

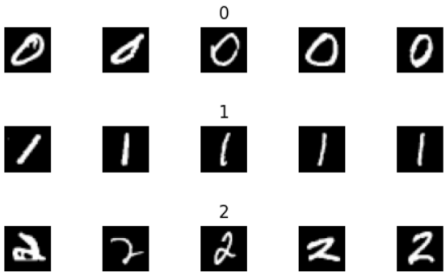
File Edit View Insert Runtime Tools Help Last edited on July 13

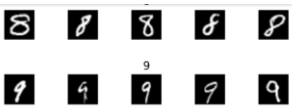
+ Code + Text

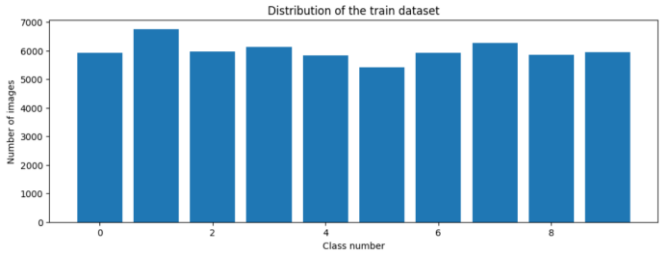
```
assert(X_test.shape[0] == y_test.shape[0]), "The number of images is not equal to the number of labels."
assert(X_test.shape[1:] == (28,28)), "The dimensions of the images are not 28 x 28."

num_of_samples=[]

cols = 5
num_classes = 10

fig, axs = plt.subplots(nrows=num_classes, ncols=cols, figsize=(5,10))
fig.tight_layout()

for i in range(cols):
    for j in range(num_classes):
      x_selected = X_train[y_train == j]
      axs[j][i].imshow(x_selected[random.randint(0,(len(x_selected) - 1)), :, :], cmap=plt.get_cmap('gray'))
      axs[j][i].axis("off")
      if i == 2:
        axs[j][i].set_title(str(j))
        num_of_samples.append(len(x_selected))



print(num_of_samples)
plt.figure(figsize=(12, 4))
plt.bar(range(0, num_classes), num_of_samples)
plt.title("Distribution of the train dataset")
plt.xlabel("Class number")
plt.ylabel("Number of images")
```

```
X_train = X_train.reshape(60000, 28, 28, 1)
X_test = X_test.reshape(10000, 28, 28, 1)


y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

X_train = X_train/255
X_test = X_test/255
```

```
(60000, 28, 28)
(10000, 28, 28)
[5923, 6742, 5958, 6131, 5842, 5421, 5918, 6265, 5851, 5949]
```








```
#define the leNet model function
def leNet_model():
  model = Sequential()
  model.add(Conv2D(30, (5, 5), input_shape = (28,28,1),  activation='relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Conv2D(15, (3, 3), activation = 'relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Flatten())
  model.add(Dense(500, activation= 'relu'))
  model.add(Dropout(0.5))
  model.add(Dense(num_classes, activation='softmax'))
  model.compile(Adam(lr=0.01), loss='categorical_crossentropy', metrics=['accuracy'])
  return model
```

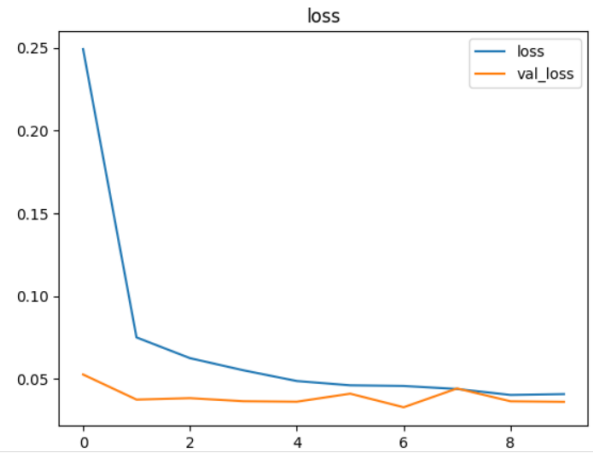
```
[ ] model = leNet_model()
    print(model.summary())
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_4 (Conv2D)           (None, 24, 24, 30)        780

 max_pooling2d_4 (MaxPooling  (None, 12, 12, 30)       0
 2D)

 conv2d_5 (Conv2D)           (None, 10, 10, 15)        4065

 max_pooling2d_5 (MaxPooling  (None, 5, 5, 15)         0
 2D)

 flatten_2 (Flatten)         (None, 375)               0

 dense_4 (Dense)             (None, 500)               188000

 dropout_1 (Dropout)         (None, 500)               0

 dense_5 (Dense)             (None, 10)                5010

=================================================================
Total params: 197,855
Trainable params: 197,855
Non-trainable params: 0
_________________________________________________________________
None
/usr/local/lib/python3.10/dist-packages/keras/optimizers/legacy/adam.py:117: UserWarning: The `lr` argument is deprecated, use `lear
  super().__init__(name, **kwargs)
```

```
history = model.fit(X_train, y_train, epochs=10, validation_split=0.1, batch_size=400, verbose=1, shuffle=1)

Epoch 1/10
135/135 [==============================] - 1s 8ms/step - loss: 0.2491 - accuracy: 0.9200 - val_loss: 0.0527 - val_accuracy: 0.9848
Epoch 2/10
135/135 [==============================] - 1s 6ms/step - loss: 0.0751 - accuracy: 0.9769 - val_loss: 0.0375 - val_accuracy: 0.9890
Epoch 3/10
135/135 [==============================] - 1s 6ms/step - loss: 0.0625 - accuracy: 0.9804 - val_loss: 0.0384 - val_accuracy: 0.9883
Epoch 4/10
135/135 [==============================] - 1s 6ms/step - loss: 0.0552 - accuracy: 0.9827 - val_loss: 0.0365 - val_accuracy: 0.9903
Epoch 5/10
135/135 [==============================] - 1s 6ms/step - loss: 0.0487 - accuracy: 0.9853 - val_loss: 0.0363 - val_accuracy: 0.9898
Epoch 6/10
135/135 [==============================] - 1s 7ms/step - loss: 0.0461 - accuracy: 0.9861 - val_loss: 0.0411 - val_accuracy: 0.9892
Epoch 7/10
135/135 [==============================] - 1s 11ms/step - loss: 0.0457 - accuracy: 0.9856 - val_loss: 0.0329 - val_accuracy: 0.9895
Epoch 8/10
135/135 [==============================] - 1s 10ms/step - loss: 0.0439 - accuracy: 0.9860 - val_loss: 0.0442 - val_accuracy: 0.9900
Epoch 9/10
135/135 [==============================] - 1s 8ms/step - loss: 0.0403 - accuracy: 0.9869 - val_loss: 0.0365 - val_accuracy: 0.9907
Epoch 10/10
135/135 [==============================] - 1s 8ms/step - loss: 0.0408 - accuracy: 0.9875 - val_loss: 0.0362 - val_accuracy: 0.9902
```

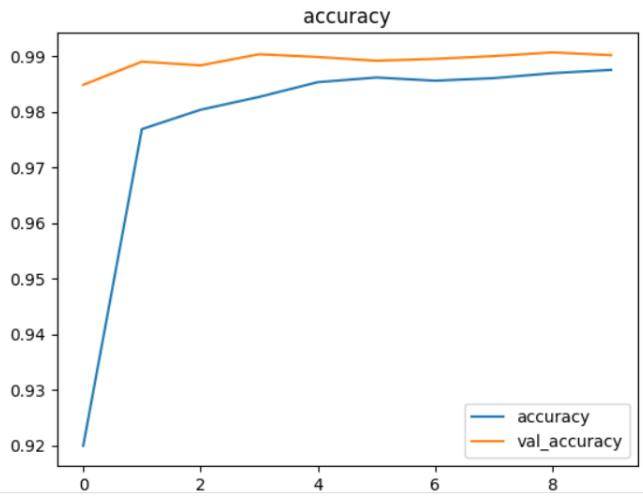
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.legend(['loss', 'val_loss'])
plt.title('loss')
plt.xlabel('epoch')
```

```
Text(0.5, 0, 'epoch')
```



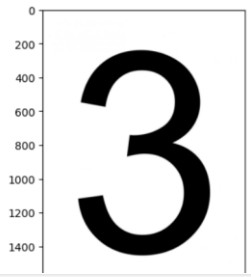

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.legend(['accuracy', 'val_accuracy'])
plt.title('accuracy')
plt.xlabel('epoch')
```

```
Text(0.5, 0, 'epoch')
```



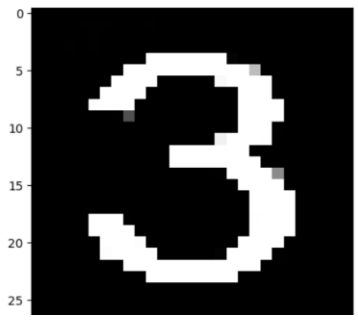

```
import requests
from PIL import Image

url = 'https://printables.space/files/uploads/download-and-print/large-printable-numbers/3-a4-1200x1697.jpg'

response = requests.get(url, stream=True)
img = Image.open(response.raw)
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x7e4768b77250>
```




```
import cv2

img_array = np.asarray(img)
resized = cv2.resize(img_array, (28,28))
gray_scale = cv2.cvtColor(resized, cv2.COLOR_BGR2GRAY)
image = cv2.bitwise_not(gray_scale)
plt.imshow(image, cmap=plt.get_cmap("gray"))
```

```
<matplotlib.image.AxesImage at 0x7e4763db88b0>
```

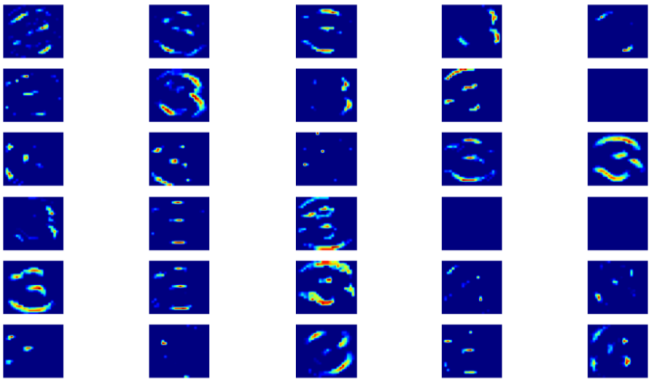


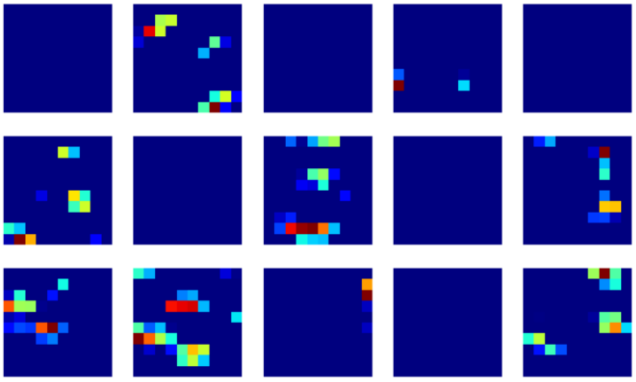

```
score = model.evaluate(X_test, y_test, verbose=0)
print(type(score))
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

```
<class 'list'>
Test score: 0.03322405740618706
Test accuracy: 0.9898999929428101
```

```
layer1 = Model(inputs=model.layers[0].input, outputs=model.layers[0].output)
layer2 = Model(inputs=model.layers[0].input, outputs=model.layers[2].output)

visual_layer1, visual_layer2 = layer1.predict(img), layer2.predict(img)

print(visual_layer1.shape)
print(visual_layer2.shape)

#layer 1
plt.figure(figsize=(10, 6))
for i in range(30):
    plt.subplot(6, 5, i+1)
    plt.imshow(visual_layer1[0, :, :, i], cmap=plt.get_cmap('jet'))
    plt.axis('off')

#layer 2
plt.figure(figsize=(10, 6))
for i in range(15):
    plt.subplot(3, 5, i+1)
    plt.imshow(visual_layer2[0, :, :, i], cmap=plt.get_cmap('jet'))
    plt.axis('off')
```

```
1/1 [==============================] - 0s 343ms/step
1/1 [==============================] - 0s 173ms/step
(1, 24, 24, 30)
(1, 10, 10, 15)
```





The above Code displays the well structured Convolutional neural network trained on number data for predicting a number like number 3 , along with the fine tuning of it for obtaining better results and scores and validation data. A well-structured convolutional neural network shows validation accuracy higher than the trained data and loss lower than the trained data, which is a specification of a well-structured neural network.

Classifying Road Symbols:

A classifying road symbol code built using CNN (Convolutional Neural Network) refers to a computer vision system that utilizes deep learning techniques, specifically CNNs, to classify and interpret road symbols found on traffic signs and road markings.

CNNs are a type of deep learning neural network that are highly effective in image recognition tasks. They are particularly well-suited for tasks like classifying road symbols because they can automatically learn important features and patterns from images, making them capable of understanding the visual information present in different road symbols.

Here's how a CNN-based system for classifying road symbols works:

Data Collection: To build a CNN-based classifier for road symbols, a large dataset of road sign images is collected. This dataset should include a wide variety of road symbols, representing different types of signs, signals, and markings commonly found on roads.

Data Preprocessing: The collected images are pre-processed to ensure consistency and improve the training process. Preprocessing steps may include resizing images to a standard size, normalizing pixel values, and using data augmentation techniques to increase the diversity of the training data.

Model Architecture: The CNN model architecture is designed to process the input images and extract relevant features from them. It typically consists of multiple layers of convolutional and pooling operations, followed by fully connected layers for classification.

Training: The CNN model is trained on the pre-processed dataset of road sign images. During training, the model adjusts its internal parameters (weights and biases) to minimize the difference between predicted and actual labels. This process is accomplished using optimization algorithms like stochastic gradient descent (SGD) or Adam.

Validation and Testing: After training, the model is evaluated using a separate validation dataset to assess its performance. Once the model meets satisfactory accuracy levels, it is tested on a test dataset to ensure generalization to new, unseen road sign images.

Classification: Once the CNN model is trained and validated, it can be used for classifying road symbols in real-world scenarios. The model takes an input image as its input and predicts the corresponding road symbol label.

The CNN-based road symbol classifier can be integrated into various applications, such as autonomous vehicles, smart city infrastructure, and road safety systems. It can help in real-time recognition of road signs and markings, assisting drivers with navigation and enhancing overall road safety. Therefore, to implement this I have developed a code to present a Representation of it in real life.

Classifying_Road_Symbols.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
[1] !git clone https://bitbucket.org/jadslim/german-traffic-signs

Cloning into 'german-traffic-signs'...
Unpacking objects: 100% (6/6), 117.80 MiB | 6.29 MiB/s, done.
Updating files: 100% (4/4), done.
```

```
[2] !ls german-traffic-sign

ls: cannot access 'german-traffic-sign': No such file or directory
```

```
[3] import numpy as np
    import matplotlib.pyplot as plt
    import keras
    from keras.models import Sequential
    from keras.optimizers import Adam
    from keras.layers import Dense
    from keras.layers import Flatten, Dropout
    from keras.utils.np_utils import to_categorical
    from keras.layers.convolutional import Conv2D, MaxPooling2D
    import random
    import pickle
    import pandas as pd
    import cv2
    from keras.callbacks import LearningRateScheduler, ModelCheckpoint

    %matplotlib inline
    np.random.seed(0)
```

Classifying_Road_Symbols.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
[4] with open('german-traffic-signs/train.p', 'rb') as f:
      train_data = pickle.load(f)
    with open('german-traffic-signs/valid.p', 'rb') as f:
      val_data = pickle.load(f)
    with open('german-traffic-signs/test.p', 'rb') as f:
      test_data = pickle.load(f)

    print(type(train_data))

    X_train, y_train = train_data['features'], train_data['labels']
    X_val, y_val = val_data['features'], val_data['labels']
    X_test, y_test = test_data['features'], test_data['labels']
```

```
<class 'dict'>
```

```
print(X_train.shape)
print(X_test.shape)
print(X_val.shape)
```

```
(34799, 32, 32, 3)
(12630, 32, 32, 3)
(4410, 32, 32, 3)
```

Classifying_Road_Symbols.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
[6] assert(X_train.shape[0] == y_train.shape[0]), "The number of images is not equal to the number of labels."
    assert(X_train.shape[1:] == (32,32,3)), "The dimensions of the images are not 32 x 32 x 3."
    assert(X_val.shape[0] == y_val.shape[0]), "The number of images is not equal to the number of labels."
    assert(X_val.shape[1:] == (32,32,3)), "The dimensions of the images are not 32 x 32 x 3."
    assert(X_test.shape[0] == y_test.shape[0]), "The number of images is not equal to the number of labels."
    assert(X_test.shape[1:] == (32,32,3)), "The dimensions of the images are not 32 x 32 x 3."
```

```
data = pd.read_csv('german-traffic-signs/signnames.csv')

num_of_samples=[]

cols = 5
num_classes = 43

fig, axs = plt.subplots(nrows=num_classes, ncols=cols, figsize=(5,50))
fig.tight_layout()

for i in range(cols):
    for j, row in data.iterrows():
      x_selected = X_train[y_train == j]
      axs[j][i].imshow(x_selected[random.randint(0,(len(x_selected) - 1)), :, :], cmap=plt.get_cmap('gray'))
      axs[j][i].axis("off")
      if i == 2:
        axs[j][i].set_title(str(j) + " - " + row["SignName"])
        num_of_samples.append(len(x_selected))
```

Classifying_Road_Symbols.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

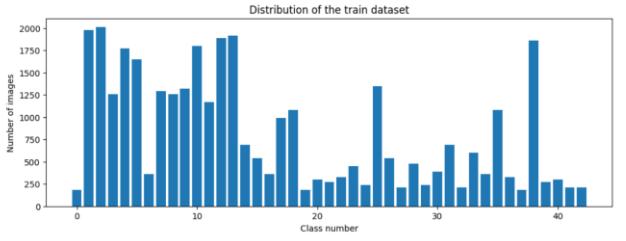
Classifying_Road_Symbols.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
[8] print(num_of_samples)
    plt.figure(figsize=(12, 4))
    plt.bar(range(0, num_classes), num_of_samples)
    plt.title("Distribution of the train dataset")
    plt.xlabel("Class number")
    plt.ylabel("Number of images")
    plt.show()
```

```
[180, 1980, 2010, 1260, 1770, 1650, 360, 1290, 1260, 1320, 1800, 1170, 1890, 1920, 690, 540, 360, 990, 1080, 180, 300, 270, 330, 450,
```



Classifying_Road_Symbols.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
import cv2
plt.imshow(X_train[1000])
plt.axis("off")
print(X_train[1000].shape)
print(y_train[1000])
```

```
(32, 32, 3)
36
```

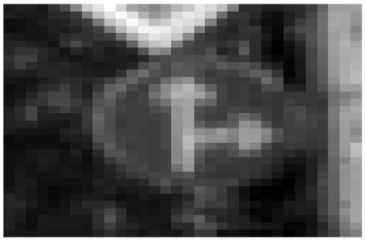
Classifying_Road_Symbols.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
def grayscale(img):
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    return img

img = grayscale(X_train[1000])
plt.imshow(img, cmap=plt.get_cmap("gray"))
plt.axis("off")
print(img.shape)
```

```
(32, 32)
```



This Code Duly Represents the Prediction of a Road symbol out of Data and the Distribution of the data set after creating rows of Road Signs.

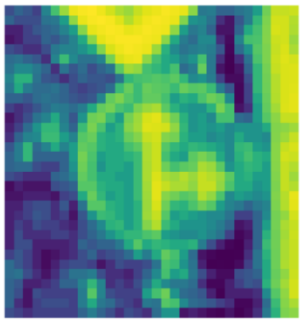
Classifying_Road_Symbols.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
[11] def equalize(img):
       img = cv2.equalizeHist(img)
       return img

     img = equalize(img)
     plt.imshow(img)
     plt.axis("off")
     print(img.shape)
```

```
(32, 32)
```



Classifying_Road_Symbols.ipynb

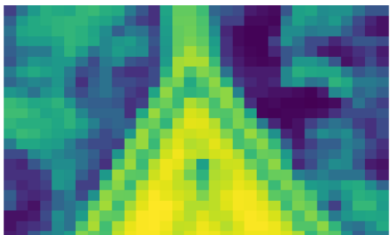
File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
def preprocessing(img):
  img = grayscale(img)
  img = equalize(img)
  img = img/255
  return img

X_train = np.array(list(map(preprocessing, X_train)))
X_val = np.array(list(map(preprocessing, X_val)))
X_test = np.array(list(map(preprocessing, X_test)))
```

```
[13] plt.imshow(X_train[random.randint(0, len(X_train)-1)])
     plt.axis("off")
     print(X_train.shape)
```

```
(34799, 32, 32)
```



Classifying_Road_Symbols.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
[17] print(X_train.shape)
     print(X_test.shape)
     print(X_val.shape)
```

```
(34799, 32, 32, 1)
(12630, 32, 32, 1)
(4410, 32, 32, 1)
```

```
y_train = to_categorical(y_train, 43)
y_test = to_categorical(y_test, 43)
y_val = to_categorical(y_val, 43)
```

```
[19] def leNet_model():
       model = Sequential()
       model.add(Conv2D(30, (5, 5), input_shape=(32, 32, 1), activation ='relu'))
       model.add(MaxPooling2D(pool_size=(2,2)))
       model.add(Conv2D(15, (3, 3), activation = 'relu'))
       model.add(MaxPooling2D(pool_size=(2, 2)))
       model.add(Flatten())
       model.add(Dense(500, activation='relu'))
       model.add(Dropout(0.5))
       model.add(Dense(num_classes, activation='softmax'))

       model.compile(Adam(lr= 0.01), loss ='categorical_crossentropy', metrics = ['accuracy'])
       return model
```

```
[20] model = leNet_model()
     print(model.summary())
```

```
Model: "sequential"
```

```
model = leNet_model()
print(model.summary())
```

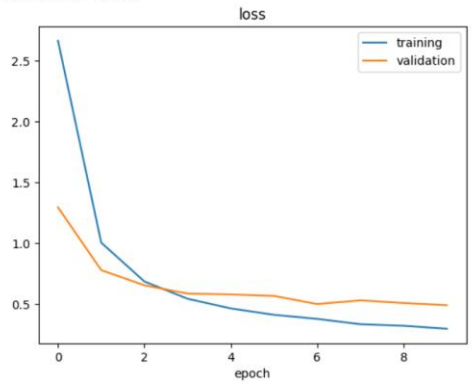
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 30)        780

 max_pooling2d (MaxPooling2D  (None, 14, 14, 30)       0
 )

 conv2d_1 (Conv2D)           (None, 12, 12, 15)        4065

 max_pooling2d_1 (MaxPooling  (None, 6, 6, 15)         0
 2D)

 flatten (Flatten)           (None, 540)               0

 dense (Dense)               (None, 500)               270500

 dropout (Dropout)           (None, 500)               0

 dense_1 (Dense)             (None, 43)                21543

=================================================================
Total params: 296,888
Trainable params: 296,888
Non-trainable params: 0
_________________________________________________________________
None
/usr/local/lib/python3.10/dist-packages/keras/optimizers/legacy/adam.py:117:
  super().__init__(name, **kwargs)
```

```
history = model.fit(X_train, y_train, epochs = 10, validation_data= (X_val, y_val), batch_size = 400, verbose = 1, shuffle = 1)
```

```
Epoch 1/10
87/87 [==============================] - 10s 13ms/step - loss: 2.6645 - accuracy: 0.2725 - val_loss: 1.2941 - val_accuracy: 0.5982
Epoch 2/10
87/87 [==============================] - 1s 7ms/step - loss: 1.0040 - accuracy: 0.6821 - val_loss: 0.7783 - val_accuracy: 0.7667
Epoch 3/10
87/87 [==============================] - 1s 7ms/step - loss: 0.6846 - accuracy: 0.7799 - val_loss: 0.6529 - val_accuracy: 0.8043
Epoch 4/10
87/87 [==============================] - 1s 7ms/step - loss: 0.5426 - accuracy: 0.8294 - val_loss: 0.5850 - val_accuracy: 0.8240
Epoch 5/10
87/87 [==============================] - 1s 7ms/step - loss: 0.4630 - accuracy: 0.8517 - val_loss: 0.5780 - val_accuracy: 0.8295
Epoch 6/10
87/87 [==============================] - 1s 7ms/step - loss: 0.4103 - accuracy: 0.8694 - val_loss: 0.5658 - val_accuracy: 0.8456
Epoch 7/10
87/87 [==============================] - 1s 7ms/step - loss: 0.3769 - accuracy: 0.8789 - val_loss: 0.4989 - val_accuracy: 0.8565
Epoch 8/10
87/87 [==============================] - 1s 7ms/step - loss: 0.3331 - accuracy: 0.8920 - val_loss: 0.5293 - val_accuracy: 0.8569
Epoch 9/10
87/87 [==============================] - 1s 7ms/step - loss: 0.3204 - accuracy: 0.8954 - val_loss: 0.5077 - val_accuracy: 0.8592
Epoch 10/10
87/87 [==============================] - 1s 8ms/step - loss: 0.2959 - accuracy: 0.9028 - val_loss: 0.4891 - val_accuracy: 0.8712
```

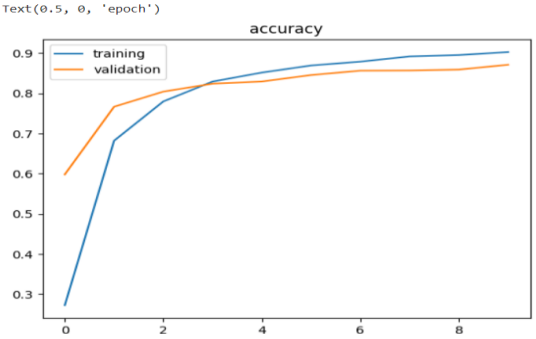
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.legend(['training', 'validation'])
plt.title('loss')
plt.xlabel('epoch')
```

```
Text(0.5, 0, 'epoch')
```




```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.legend(['training', 'validation'])
plt.title('accuracy')
plt.xlabel('epoch')
```

```
Text(0.5, 0, 'epoch')
```



Now after analysing the code, we have to improve this for getting better accuracy in the model now we will do fine tuning of the model by modifying the model and adding dropout layers.

```
[42] def modified_model(): # fine tuning again
       model = Sequential()
       model.add(Conv2D(30, (5, 5), input_shape=(32, 32, 1), activation ='relu'))
       model.add(Conv2D(30, (5, 5), activation ='relu'))
       model.add(MaxPooling2D(pool_size=(2,2)))

       model.add(Conv2D(15, (3, 3), activation = 'relu'))
       model.add(Conv2D(15, (3, 3), activation = 'relu'))
       model.add(MaxPooling2D(pool_size=(2, 2)))
       model.add(Dropout(0.5))

       model.add(Flatten())
       model.add(Dense(500, activation='relu'))
       model.add(Dropout(0.5))
       model.add(Dense(num_classes, activation='softmax'))

       model.compile(Adam(lr= 0.001), loss ='categorical_crossentropy', metrics = ['accuracy'])
       return model
```

```
[43] model = modified_model()
     print(model.summary())
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_10 (Conv2D)          (None, 28, 28, 30)        780

 conv2d_11 (Conv2D)          (None, 24, 24, 30)        22530

 max_pooling2d_8 (MaxPooling  (None, 12, 12, 30)       0
 2D)
```

```
model = modified_model()
print(model.summary())
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_10 (Conv2D)          (None, 28, 28, 30)        780

 conv2d_11 (Conv2D)          (None, 24, 24, 30)        22530

 max_pooling2d_8 (MaxPooling  (None, 12, 12, 30)       0
 2D)

 conv2d_12 (Conv2D)          (None, 10, 10, 15)        4065

 conv2d_13 (Conv2D)          (None, 8, 8, 15)          2040

 max_pooling2d_9 (MaxPooling  (None, 4, 4, 15)         0
 2D)

 dropout_4 (Dropout)         (None, 4, 4, 15)          0

 flatten_4 (Flatten)         (None, 240)               0

 dense_8 (Dense)             (None, 500)               120500

 dropout_5 (Dropout)         (None, 500)               0

 dense_9 (Dense)             (None, 43)                21543

=================================================================
Total params: 171,458
Trainable params: 171,458
Non-trainable params: 0
```

```
history = model.fit(X_train, y_train, epochs = 10, validation_data= (X_val, y_val), batch_size = 400, verbose = 1, shuffle = 1)
```

```
Epoch 1/10
87/87 [==============================] - 3s 22ms/step - loss: 2.8014 - accuracy: 0.2516 - val_loss: 1.1423 - val_accuracy: 0.6796
Epoch 2/10
87/87 [==============================] - 1s 16ms/step - loss: 1.0668 - accuracy: 0.6729 - val_loss: 0.4979 - val_accuracy: 0.8658
Epoch 3/10
87/87 [==============================] - 1s 16ms/step - loss: 0.6910 - accuracy: 0.7852 - val_loss: 0.3263 - val_accuracy: 0.9034
Epoch 4/10
87/87 [==============================] - 1s 16ms/step - loss: 0.5354 - accuracy: 0.8328 - val_loss: 0.2788 - val_accuracy: 0.9179
Epoch 5/10
87/87 [==============================] - 1s 16ms/step - loss: 0.4476 - accuracy: 0.8588 - val_loss: 0.2133 - val_accuracy: 0.9392
Epoch 6/10
87/87 [==============================] - 1s 16ms/step - loss: 0.3856 - accuracy: 0.8770 - val_loss: 0.1811 - val_accuracy: 0.9474
Epoch 7/10
87/87 [==============================] - 1s 16ms/step - loss: 0.3456 - accuracy: 0.8907 - val_loss: 0.1587 - val_accuracy: 0.9578
Epoch 8/10
87/87 [==============================] - 1s 16ms/step - loss: 0.3104 - accuracy: 0.9015 - val_loss: 0.1894 - val_accuracy: 0.9413
Epoch 9/10
87/87 [==============================] - 2s 18ms/step - loss: 0.2829 - accuracy: 0.9107 - val_loss: 0.1368 - val_accuracy: 0.9603
Epoch 10/10
87/87 [==============================] - 1s 17ms/step - loss: 0.2563 - accuracy: 0.9192 - val_loss: 0.1288 - val_accuracy: 0.9680
```

```
score = model.evaluate(X_test, y_test, verbose = 0)
print('Test Score:', score[0])
```

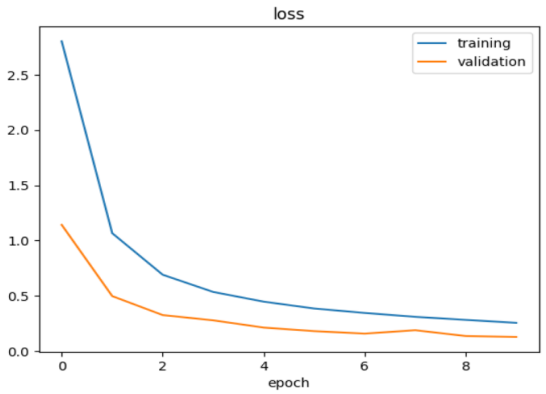
```
Test Score: 0.20648494362831116
```

```
score = model.evaluate(X_test, y_test, verbose = 1)
print('Test Score:', score[1])
```

```
395/395 [==============================] - 1s 3ms/step - loss: 0.2065 - accuracy: 0.9410
Test Score: 0.9410134553909302
```

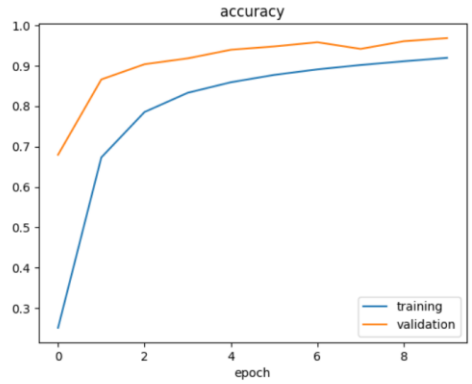
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.legend(['training', 'validation'])
plt.title('loss')
plt.xlabel('epoch')
```

```
Text(0.5, 0, 'epoch')
```




```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.legend(['training', 'validation'])
plt.title('accuracy')
plt.xlabel('epoch')
```

```
Text(0.5, 0, 'epoch')
```



After Fine Tuning the model and modifying the model we get a better score and accuracy, the technique of fit generator helps the process and model a lot in achieving a good and better score while training a machine learning model.

```
#Reshape reshape

img = img.reshape(1, 32, 32, 1)
```

```
#Fit_generator
history = model.fit_generator(datagen.flow(X_train, y_train, batch_size=50), steps_per_epoch=2000, epochs=10, validation_data=(X_val, y_val),shuffle = 1)
```

```
Epoch 1/10
<ipython-input-64-f5de16b2c030>:2: UserWarning: `Model.fit_generator` is deprecated and will be removed in a future version. Please use `Model.fit`, which
  history = model.fit_generator(datagen.flow(X_train, y_train, batch_size=50), steps_per_epoch=2000, epochs=10, validation_data=(X_val, y_val),shuffle = 1
 694/2000 [=========>....................] - ETA: 28s - loss: 1.2409 - accuracy: 0.6335WARNING:tensorflow:Your input ran out of data; interrupting trainin
2000/2000 [==============================] - 17s 8ms/step - loss: 1.2404 - accuracy: 0.6337 - val_loss: 0.2391 - val_accuracy: 0.9465
```

```
#Fit_generator
history = model.fit(datagen.flow(X_train, y_train, batch_size=50), steps_per_epoch=2000, epochs=10, validation_data=(X_val, y_val),shuffle = 1)
```

```
Epoch 1/10
 695/2000 [=========>....................] - ETA: 28s - loss: 0.8478 - accuracy: 0.7378WARNING:tensorflow:Your input ran out of data; interrupting trainin
2000/2000 [==============================] - 16s 8ms/step - loss: 0.8474 - accuracy: 0.7380 - val_loss: 0.1756 - val_accuracy: 0.9585
```

Behavioural Cloning :

Behavioural cloning, in the context of the Udacity Car simulator, is a technique used to train a self-driving car model using data collected from human driving behaviour. The idea is to mimic the behaviour of a human driver by training a neural network to predict steering angles and control the car's movements based on input images captured from the car's front-facing camera.

Here's an overview of how behavioural cloning is implemented using the Udacity Car simulator:

Data Collection: First, human drivers operate the car in the simulator while the simulator records their driving behaviour. During this process, the simulator captures images from the car's camera and records the corresponding steering angles, throttle, brake, and speed information.

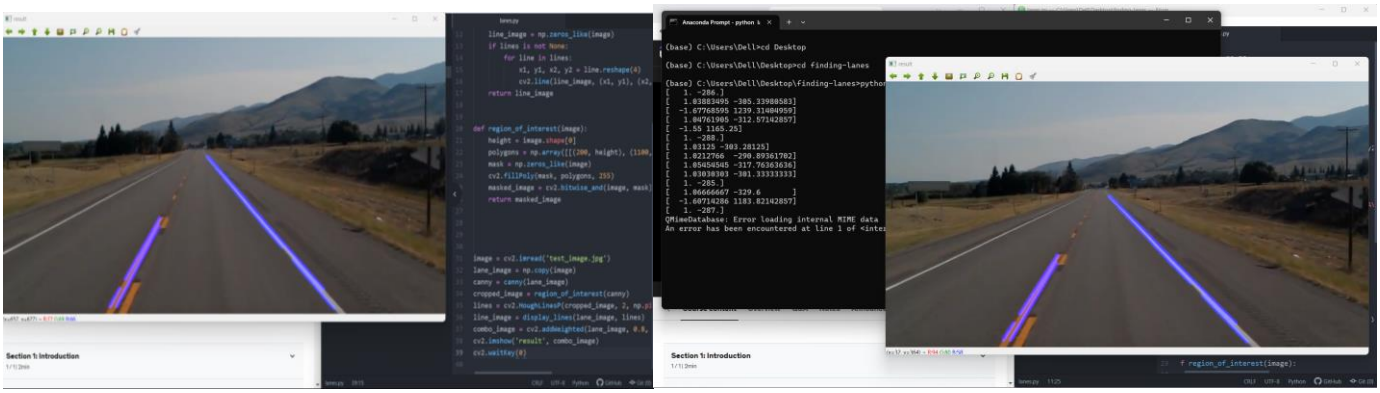
Data Preprocessing: The recorded data is then pre-processed to prepare it for training. Image data is usually resized and normalized to ensure consistency, and steering angles are adjusted to take into account potential corrections made by the human drivers.

Neural Network Architecture: A convolutional neural network (CNN) is commonly used for behavioural cloning. The CNN is designed to take in the pre-processed images as input and predict the steering angle as output. The architecture of the CNN can vary, but it typically consists of several convolutional layers to extract features from the images, followed by fully connected layers to predict the steering angle.

Training: The pre-processed data is split into training and validation sets. The CNN is then trained using the training data, and its performance is evaluated on the validation set. During training, the neural network's weights are adjusted iteratively to minimize the difference between the predicted steering angles and the actual steering angles recorded during human driving.

Validation: The model's performance is regularly monitored using the validation set. The goal is to achieve a low validation loss, indicating that the model is generalizing well to unseen data and not overfitting the training data.

Testing: Once the CNN model is trained and validated, it is tested in the simulator under autonomous mode. The model takes the real-time camera feed as input, processes it through the CNN, and predicts the steering angle to control the car's steering, mimicking human driving behaviour. Let’s Dive into the depth of Computer vision and code to know it better and it plays a vital role in working on autonomous car system as before implementing in real life we should focus on making our model more precise and up to the mark accurate.



Behavioural_Cloning.ipynb

File Edit View Insert Runtime Tools Help Last edited on July 15

+ Code + Text

```
!git clone https://github.com/AdityaK210901/Track1.git
```

```
fatal: destination path 'Track1' already exists and is not an empty directory.
```

```
[ ] !ls Track1
```

```
driving_log.csv  IMG
```

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import keras
from keras.models import Sequential
from keras.optimizers import Adam
from keras.layers import Convolution2D, MaxPooling2D, Dropout, Flatten, Dense
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
import cv2
import pandas as pd
import ntpath
import random
import os
```

```
[ ] datadir= 'Track1'
columns = ['center', 'left', 'right', 'steering', 'throttle', 'reverse', 'speed']
data = pd.read_csv(os.path.join(datadir, 'driving_log.csv'), names = columns)
pd.set_option('display.max_colwidth', -1)
data.head()
```

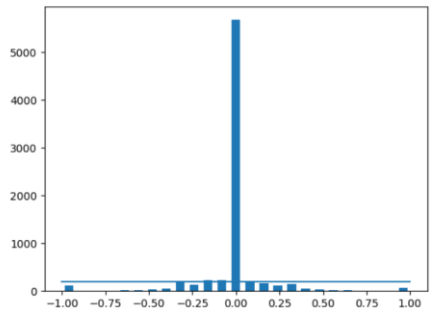
	center	left	right	steering	throttle	reverse	speed
0	C:\Simulator_Data\IMG\center_2023_07_15_13_35_03_170.jpg	C:\Simulator_Data\IMG\left_2023_07_15_13_35_03_170.jpg	C:\Simulator_Data\IMG\right_2023_07_15_13_35_03_170.jpg	0.0	0.0	0.0	0.000078
1	C:\Simulator_Data\IMG\center_2023_07_15_13_35_03_285.jpg	C:\Simulator_Data\IMG\left_2023_07_15_13_35_03_285.jpg	C:\Simulator_Data\IMG\right_2023_07_15_13_35_03_285.jpg	0.0	0.0	0.0	0.000079
2	C:\Simulator_Data\IMG\center_2023_07_15_13_35_03_386.jpg	C:\Simulator_Data\IMG\left_2023_07_15_13_35_03_386.jpg	C:\Simulator_Data\IMG\right_2023_07_15_13_35_03_386.jpg	0.0	0.0	0.0	0.000082
3	C:\Simulator_Data\IMG\center_2023_07_15_13_35_03_489.jpg	C:\Simulator_Data\IMG\left_2023_07_15_13_35_03_489.jpg	C:\Simulator_Data\IMG\right_2023_07_15_13_35_03_489.jpg	0.0	0.0	0.0	0.000078
4	C:\Simulator_Data\IMG\center_2023_07_15_13_35_03_591.jpg	C:\Simulator_Data\IMG\left_2023_07_15_13_35_03_591.jpg	C:\Simulator_Data\IMG\right_2023_07_15_13_35_03_591.jpg	0.0	0.0	0.0	0.000080

```
def path_leaf(path):
  head, tail = ntpath.split(path)
  return tail
data['center'] = data['center'].apply(path_leaf)
data['left'] = data['left'].apply(path_leaf)
data['right'] = data['right'].apply(path_leaf)
data.head()
```

	center	left	right	steering	throttle	reverse	speed
0	center_2023_07_15_13_35_03_170.jpg	left_2023_07_15_13_35_03_170.jpg	right_2023_07_15_13_35_03_170.jpg	0.0	0.0	0.0	0.000078
1	center_2023_07_15_13_35_03_285.jpg	left_2023_07_15_13_35_03_285.jpg	right_2023_07_15_13_35_03_285.jpg	0.0	0.0	0.0	0.000079
2	center_2023_07_15_13_35_03_386.jpg	left_2023_07_15_13_35_03_386.jpg	right_2023_07_15_13_35_03_386.jpg	0.0	0.0	0.0	0.000082
3	center_2023_07_15_13_35_03_489.jpg	left_2023_07_15_13_35_03_489.jpg	right_2023_07_15_13_35_03_489.jpg	0.0	0.0	0.0	0.000078
4	center_2023_07_15_13_35_03_591.jpg	left_2023_07_15_13_35_03_591.jpg	right_2023_07_15_13_35_03_591.jpg	0.0	0.0	0.0	0.000080

```
num_bins = 25
samples_per_bin = 200
hist, bins = np.histogram(data['steering'], num_bins)
center = (bins[:-1]+ bins[1:]) * 0.5
plt.bar(center, hist, width=0.05)
plt.plot((np.min(data['steering']), np.max(data['steering'])), (samples_per_bin, samples_per_bin))
print(bins)
```

```
[-1.   -0.92 -0.84 -0.76 -0.68 -0.6  -0.52 -0.44 -0.36 -0.28 -0.2  -0.12
 -0.04  0.04  0.12  0.2   0.28  0.36  0.44  0.52  0.6   0.68  0.76  0.84
  0.92  1.  ]
```



Behavioural_Cloning.ipynb

File Edit View Insert Runtime Tools Help Last edited on July 15

+ Code + Text

```
# Now balancing the data#
```

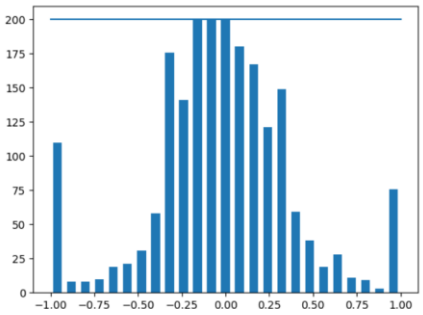
```
[ ] print('total data:', len(data))
    remove_list = []
    for j in range(num_bins):
      list_ = []
      for i in range(len(data['steering'])):
        if data['steering'][i] >= bins[j] and data['steering'][i] <= bins[j+1]:
          list_.append(i)
      list_ = shuffle(list_)
      list_ = list_[samples_per_bin:]
      remove_list.extend(list_)

    print('removed:', len(remove_list))
    data.drop(data.index[remove_list], inplace=True)
    print('remaining:', len(data))

    hist, _ = np.histogram(data['steering'], (num_bins))
    plt.bar(center, hist, width=0.05)
    plt.plot((np.min(data['steering']), np.max(data['steering'])), (samples_per_bin, samples_per_bin))
```

Behavioural_Cloning.ipynb

File Edit View Insert Runtime Tools Help Last edited on July 15

+ Code + Text

```
total data: 7566
removed: 5524
remaining: 2042
[<matplotlib.lines.Line2D at 0x79d690571ff0>]
```

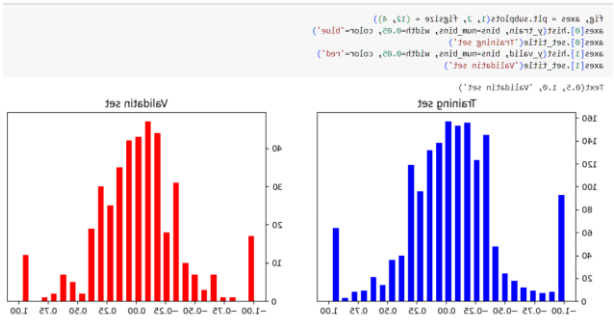



```
print(data.iloc[1])
def load_img_steering(datadir, df):
  image_path = []
  steering = []
  for i in range(len(data)):
    indexed_data = data.iloc[i]
    center, left, right = indexed_data[0], indexed_data[1], indexed_data[2]

    image_path.append(os.path.join(datadir, center.strip()))
    steering.append(float(indexed_data[3]))

  image_paths = np.asarray(image_path)
  steerings = np.asarray(steering)
  return image_paths, steerings

image_paths, steerings = load_img_steering(datadir + '/IMG', data)
```

```
center      center_2023_07_15_13_35_06_568.jpg
left        left_2023_07_15_13_35_06_568.jpg
right       right_2023_07_15_13_35_06_568.jpg
steering    0.0
throttle    0.0
reverse     0.0
speed       0.000079
Name: 32, dtype: object
```

```
[ ] X_train, X_valid,y_train, y_valid = train_test_split(image_paths, steerings, test_size=0.2, random_state =6)
    print('Training Samples: {}\nValid Samples; {}'.format(len(X_train), len(X_valid)))
```

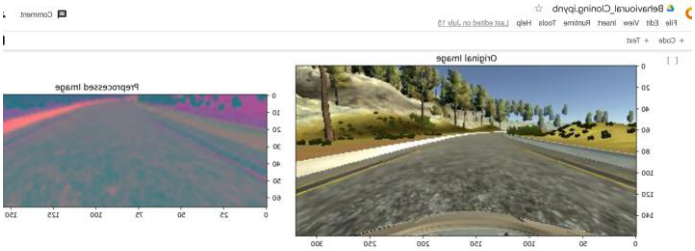
```
Training Samples: 1633
Valid Samples; 409
```




```
#preprocessing

def img_preprocess(img):
  img = mpimg.imread(img)
  img = img[60:135, :,:]
  img = cv2.cvtColor(img, cv2.COLOR_RGB2YUV)
  img = cv2.GaussianBlur(img, (3,3), 0)
  img = cv2.resize(img, (200, 66))
  img = img/255
  return img
```

```
image  = image_paths[500]
original_image = mpimg.imread(image)
preprocessed_image = img_preprocess(image)

fig, axs = plt.subplots(1, 2, figsize=(15, 10))
fig.tight_layout()
axs[0].imshow(original_image)
axs[0].set_title('Original Image')
axs[1].imshow(preprocessed_image)
axs[1].set_title('Preprocessed Image')
```

```
Text(0.5, 1.0, 'Preprocessed Image')
```




```
plt.imshow(X_train[random.randint(0, len(X_train)-1)])
plt.axis('off')
print(X_train.shape)
```

```
(1633, 66, 200, 3)
```

```
def nvidia_model():
  model = Sequential()
  model.add(Convolution2D(24, kernel_size=(5,5), strides=(2,2), input_shape=(66,200,3),activation='relu'))
  model.add(Convolution2D(36, kernel_size=(5,5), strides=(2,2), activation='elu'))
  model.add(Convolution2D(48, kernel_size=(5,5), strides=(2,2), activation='elu'))
  model.add(Convolution2D(64, kernel_size=(3,3), activation='elu'))
  model.add(Convolution2D(64, kernel_size=(3,3), activation='elu'))
  model.add(Dropout(0.5))

  model.add(Flatten())
  model.add(Dense(100, activation='elu'))
  model.add(Dropout(0.5))

  model.add(Dense(50, activation='elu'))
  model.add(Dropout(0.5))
  model.add(Dense(10, activation ='elu'))
  model.add(Dense(1))

  optimizer= Adam(learning_rate=1e-3)
  model.compile(loss='mse', optimizer=optimizer)

  return model
```

```
dense_29 (Dense)            (None, 50)                5050

dropout_16 (Dropout)        (None, 50)                0

dense_30 (Dense)            (None, 10)                510

dropout_17 (Dropout)        (None, 10)                0

dense_31 (Dense)            (None, 1)                 11

=================================================================
Total params: 252,219
Trainable params: 252,219
Non-trainable params: 0
_________________________________________________________________
None
```

```
model = nvidia_model()
print(model.summary())
```

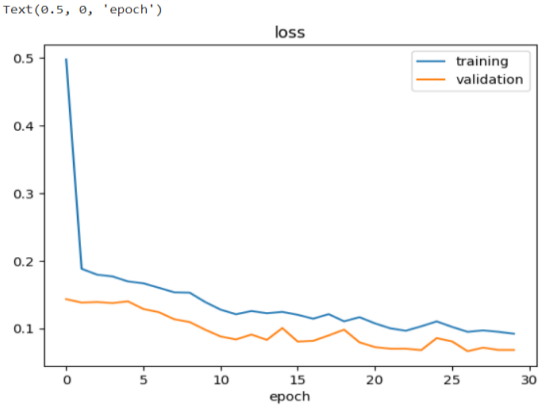
```
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_35 (Conv2D)           (None, 31, 98, 24)        1824

conv2d_36 (Conv2D)           (None, 14, 47, 36)        21636

conv2d_37 (Conv2D)           (None, 5, 22, 48)         43248

conv2d_38 (Conv2D)           (None, 3, 20, 64)         27712

conv2d_39 (Conv2D)           (None, 1, 18, 64)         36928

dropout_14 (Dropout)         (None, 1, 18, 64)         0

flatten_7 (Flatten)          (None, 1152)              0

dense_28 (Dense)             (None, 100)               115300

dropout_15 (Dropout)         (None, 100)               0
```

```
history = model.fit(X_train, y_train, epochs = 30, validation_data = (X_valid, y_valid), batch_size=100, verbose=1, shuffle=1 )
```

```
Epoch 2/30
17/17 [==============================] - 1s 30ms/step - loss: 0.1883 - val_loss: 0.1383
Epoch 3/30
17/17 [==============================] - 1s 30ms/step - loss: 0.1796 - val_loss: 0.1391
Epoch 4/30
17/17 [==============================] - 0s 30ms/step - loss: 0.1770 - val_loss: 0.1376
Epoch 5/30
17/17 [==============================] - 0s 26ms/step - loss: 0.1696 - val_loss: 0.1401
Epoch 6/30
17/17 [==============================] - 0s 24ms/step - loss: 0.1668 - val_loss: 0.1288
Epoch 7/30
17/17 [==============================] - 0s 24ms/step - loss: 0.1602 - val_loss: 0.1241
Epoch 8/30
17/17 [==============================] - 0s 26ms/step - loss: 0.1535 - val_loss: 0.1137
Epoch 9/30
17/17 [==============================] - 0s 27ms/step - loss: 0.1530 - val_loss: 0.1095
Epoch 14/30
17/17 [==============================] - 0s 26ms/step - loss: 0.1226 - val_loss: 0.0832
Epoch 15/30
17/17 [==============================] - 0s 24ms/step - loss: 0.1245 - val_loss: 0.1006
Epoch 16/30
17/17 [==============================] - 0s 25ms/step - loss: 0.1202 - val_loss: 0.0806
Epoch 17/30
17/17 [==============================] - 0s 25ms/step - loss: 0.1145 - val_loss: 0.0818
Epoch 18/30
17/17 [==============================] - 0s 24ms/step - loss: 0.1212 - val_loss: 0.0897
Epoch 19/30
17/17 [==============================] - 0s 25ms/step - loss: 0.1106 - val_loss: 0.0983
Epoch 20/30
17/17 [==============================] - 0s 25ms/step - loss: 0.1167 - val_loss: 0.0795
Epoch 21/30
17/17 [==============================] - 0s 25ms/step - loss: 0.1076 - val_loss: 0.0725
Epoch 22/30
17/17 [==============================] - 0s 25ms/step - loss: 0.1002 - val_loss: 0.0701
Epoch 23/30
17/17 [==============================] - 0s 25ms/step - loss: 0.0967 - val_loss: 0.0701
Epoch 24/30
17/17 [==============================] - 0s 25ms/step - loss: 0.1032 - val_loss: 0.0681
Epoch 25/30
17/17 [==============================] - 0s 26ms/step - loss: 0.1105 - val_loss: 0.0858
Epoch 26/30
17/17 [==============================] - 0s 25ms/step - loss: 0.1023 - val_loss: 0.0808
Epoch 27/30
17/17 [==============================] - 0s 25ms/step - loss: 0.0951 - val_loss: 0.0664
Epoch 28/30
17/17 [==============================] - 0s 27ms/step - loss: 0.0972 - val_loss: 0.0717
Epoch 29/30
17/17 [==============================] - 1s 33ms/step - loss: 0.0951 - val_loss: 0.0683
Epoch 30/30
17/17 [==============================] - 1s 30ms/step - loss: 0.0923 - val_loss: 0.0683
```

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.legend(['training', 'validation'])
plt.title('loss')
plt.xlabel('epoch')
```

```
Text(0.5, 0, 'epoch')
```




```
[ ]  model.save('model.h5')
```

```
from google.colab import files
files.download('model.h5')
```

In conclusion, this work report highlights the successful implementation of two vital components in the field of autonomous driving: classifying road symbols and behavioural cloning. The development of a robust Convolutional Neural Network (CNN) for classifying road symbols has enabled accurate recognition of traffic signs and road markings, crucial for safe and efficient navigation. Additionally, the application of behavioural cloning using the Udacity Car simulator has demonstrated the ability to mimic human driving behaviour, facilitating the creation of self-driving car models capable of navigating complex road environments. By combining these advancements, we have taken significant strides towards the realization of safer, more reliable autonomous vehicles. However, continued research, real-world testing, and optimization are essential to further enhance the performance and adaptability of these models in real-life driving scenarios. With these advancements, the vision of a future with autonomous vehicles contributing to improved road safety and transportation efficiency becomes more attainable.